An Abstract and Reusable Programming Language Architecture

Hassan Aït-Kaci
hak@ilog.fr

Ilog
Research and Development
Optimization Group
http://www.ilog.fr

9, rue de Verdun - B.P. 85
94253 Gentilly Cedex, France

December 8, 2006

(Incomplete Draft)

Preamble

Purpose

The purpose of this document is to describe, explain, and justify the design of the `ilog.language.design` package. Its main goal is to serve as a specification as well as a documentation of the details of various of its intricacies. As such, it serves mainly its author helping him to keep track of subtleties he alone may know of but may not remember—at least not in full detail—and, of course, it is also meant for those brave enough to use it, let alone those who wish to understand it in details (*gasp!*) to adapt and/or extend its functionalities.

Acknowledgement

Many thanks are due to the few, the proud, the *"volunteer"* pre-α-testers of the viability of the whole design—especially the NGO design team,[1] and any others having been exposed, willy or nilly, to some of, or the whole package as its design was unfolding!...

Many thanks also to ILOG for their open mind, as well as their keen acumen making the savvy intellectual investment of trusting their R&D to make no compromise in the best quality of their software. I have enjoyed the challenge of meeting their customers demands with the best possible scientific environment.[2]

[1] Thank you Chritiane Bracchi, Chrisptophe Gefflot, Frédéric Paulin, and Patrick Viry!... ;-)

[2] Thanks, in particular, to Jean-Fran⌋ois Puget, ILOG's VP of Optimization, and Jean-Fran⌋ois Abramatic, ILOG's CTO, for their support, and of course Pierre Haren, ILOG's CEO for his tireless contagious enthusiasm.

Contents

1 Programming language design **1**

2 Overview **3**

2.1 Abstract programming language design 3

2.1.1 Surface language 3

2.1.2 Kernel language 3

2.1.3 Type language 3

2.1.4 Intermediate language 3

2.1.5 Execution backend 3

2.1.6 Pragmatics 3

3 The kernel language **5**

3.1 Kernel expression 5

3.2 Processing a kernel expression 5

3.2.1 Sanitizer 6

3.2.2 Typechecker 6

3.2.3 Compiler 6

3.3 Description of kernel expressions 7

3.3.1 Constant 11

3.3.2 Abstraction 12

3.3.3 Application 13

3.3.4 Local 14

3.3.5 Parameter 14

3.3.6 Global . . . 14
3.3.7 Dummy . . . 14
3.3.8 Definition . . . 14
3.3.9 IfThenElse . . . 14
3.3.10 AndOr . . . 14
3.3.11 Sequence . . . 16
3.3.12 Let . . . 16
3.3.13 Loop . . . 16
3.3.14 ExitWithValue . . . 17
3.3.15 Assignment . . . 19
3.3.16 NewArray . . . 19
3.3.17 ArraySlot . . . 19
3.3.18 ArraySlotUpdate . . . 19
3.3.19 ArrayExtension . . . 19
3.3.20 ArrayInitializer . . . 19
3.3.21 Tuple . . . 19
3.3.22 NamedTuple . . . 19
3.3.23 TupleProjection . . . 19
3.3.24 TupleUpdate . . . 19
3.3.25 NewObject . . . 19
3.3.26 DottedNotation . . . 19
3.3.27 FieldUpdate . . . 21
3.3.28 Homomorphism . . . 21
3.3.29 Comprehension . . . 24

4 The type language 49
4.1 Overview . . . 49
4.1.1 Polymorphism . . . 49
4.1.2 Multiple Type Overloading . . . 50
4.1.3 Currying . . . 50
4.1.4 Standardizing . . . 50

4.1.5 Copying . . . 51
4.1.6 Equality . . . 51
4.1.7 Unifying . . . 51
4.1.8 Boxing/Unboxing . . . 51
4.2 The type system . . . 52
4.3 Static types . . . 53
4.3.1 Primitive types . . . 53
4.3.2 Type constructors . . . 53
4.3.3 Polymorphic types . . . 56
4.4 Type definitions . . . 56
4.4.1 Type aliasing . . . 57
4.4.2 Type hiding . . . 57
4.5 Dynamic types . . . 57
4.5.1 Extensional types . . . 58
4.5.2 Intensional types . . . 58

5 The intermediate language 59
5.1 Do-nothing instruction . . . 59
5.2 Push instructions . . . 59
5.3 Subroutine instructions . . . 60
5.4 Pop instructions . . . 61
5.5 Relocatable instructions . . . 61
5.6 Conversion instructions . . . 61
5.7 Assignment instructions . . . 62
5.8 Tuple component instructions . . . 62
5.9 Array/Map allocation instructions . . . 62
5.10 Array/Map slot instructions . . . 63
5.11 Field instructions . . . 64
5.12 Built-in operations . . . 64
5.12.1 Arithmetic operations . . . 64
5.12.2 Arithmetic relations . . . 65

5.12.3 Boolean operations . . . 66
5.12.4 Map and Size operations . . . 66
5.12.5 Container operations . . . 66
5.12.6 Set operations . . . 67
5.12.7 Set relations . . . 67
5.12.8 Set element operations . . . 67
5.12.9 Range operations . . . 68
5.12.10 String operations . . . 68
5.12.11 I/O operations . . . 68
5.13 Dummy instructions . . . 69

6 The backend system 71
6.1 The runtime system . . . 71
6.2 The runtime objects . . . 72
6.3 The display manager . . . 72
6.4 The error manager . . . 72

7 A full example—HAK_LL 73

8 Conclusion 75

A A word on traceability 77
A.1 Relating concrete and abstract syntax . . . 77
A.1.1 Syntax errors . . . 77
A.1.2 Static Semantics errors . . . 77
A.1.3 Dynamic Semantics errors . . . 77
A.2 Displaying and reading . . . 77
A.2.1 Displaying . . . 78
A.2.2 Reading . . . 78
A.2.3 Concretizing abstract syntax down . . . 78
A.2.4 Abstracting concrete syntax away . . . 78

B A four-panelled architecture 79

B.1 The Complete Kernel 79
B.1.1 Sanitizing 79
B.1.2 Type checking *vs.* inference 79
B.1.3 Compiling 79
B.2 The Complete Type System 80
B.2.1 The type prover 81
B.3 Structure of the `TypeChecker` 81
B.3.1 The type constructs 82
B.3.2 Defining new types 82
B.4 The Basic Instruction Set 82
B.5 The Complete Backend 82
B.5.1 The `Runtime` class 82
B.5.2 The `RuntimeObject` class 82
B.5.3 The `DisplayManager` class 82
B.5.4 The `ErrorManager` class 82

Chapter 1

Programming language design

Language is a means of communication. By this definition, a particular language serves as a conduit for information exchange between communicating entities. Such entities may be of various kinds (be it sentient—*e.g.*, humans, animals—or pragmatic tools—*e.g.*, elevators, cars, computers, *etc.*). A *programming language* is a language for human-to-computer or computer-to-computer communication.

A *natural language*, such as the one you are reading and I am using right now, is a language for human-to-human communication. Such languages are (generally) not designed—they *evolve*.[1] *Programming languages are designed.* They are designed today more formally thanks to linguistic research that led to syntactic science (leading to parser technology) and research in the formal denotational semantics of programming constructs. As in the case of a natural language, a grammar will regulate the formation of sentences (programs) that will be understood (interpreted/executed) according to the language's natural (denotational/operational) semantics.

Designing a programming language is difficult because it requires being aware of all the (overwhelmingly numerous) consequences of the slightest design decision that may occur anytime during the lexical or syntactical analyses, and the static or dynamic semantics phases. Because of the potentially high design costs (in time and effort, but also in terms of the quality of the end product—*viz.*, performance and reliability of the language being designed)[2] investing in defining and implementing a new language is prohibitive.

Fortunately, there have been design tools to help in the process. So-called meta-compilers have been used to great benefit and higher quality of language implementations. The "meta" part is

[1] In fact, natural languages have *co*-evolved under one another's historical and geographical influence and often mutate through exchange of syntax and/or semantics—we do not include here *artificial* human languages like *Esperanto, etc.*., ... The "damage" incurred by a language adopting a new often mutated concept from another being this co-evolution. There also are some rare natural languages that evolved away from others due to mere geographical reasons.

[2] Not to mention how to justify, let alone guarantee, the correctness of the design's implementation.

actually mostly true for the lexical and syntactic phases of the language design. Even then, the metasyntactic tools are often restricted to specific classes of grammars and/or parsing algorithms. Still fewer propose tools for *abstract syntax*. Most that do confine the abstract syntax language to some form of idiosyncratic representation of a tree language with some *ad hoc* interpretation. Even rarer are language design systems that propose abstract and reusable components in the form of expressions of a formal typed kernel calculus. This is what this work proposes, and this document explains such a design.

This work is therefore a metadesign: it is the design of a design tool. The emphasis—the novelty of what is proposed here—is not so much on the lexical/syntactical phases, but mostly on the semantic phases.[3]

This document describes the design of an abstract reusable programming language architecture and its implementation in Java. It represents the basis insofar as these abstract and reusable constructs, and any well-typed compositions thereof, may be instantiated in various modular language configurations.[4]

[3]The lexical/syntactic phases also deserve attention, and I have implemented a set of extensions to the conventional `lex/yacc` (alternatively, `flex/bison`) meta[lex/syntact]ical tools [3]. More has to be done on that side—*e.g.*, documentation!—and much of it is operational and can be used as a *de facto* [lex/syntact]ical front end to the semantic architecture proposed here.

[4]The first facet was the elaboration of Jacc, an advanced system for syntax-directed compiler generation [3]. The third facet will be the integration of logic-relational (from Logic Programming) and objet-relational (from Database Programming). A later facet may be to complete the design to enable both LIFE-technology [2] and CSP/LP technology to cohabit.

Chapter 2

Overview

2.1 Abstract programming language design

2.1.1 Surface language

2.1.2 Kernel language

2.1.3 Type language

2.1.4 Intermediate language

2.1.5 Execution backend

Semantic language: Runtime objects

Type-directed Display manager

Type-directed Data Reader

2.1.6 Pragmatics

Concrete *vs.* abstract error handling

Concrete *vs.* abstract Vocabulary

Chapter 3

The kernel language

3.1 Kernel expression

3.2 Processing a kernel expression

Typically, upon being read, an `Expression` will be:

1. *"name-sanitized"*—in the context of a `Sanitizer` to discriminate between local names and global names, and establish pointers from the local variable occurrences to the abstraction that introduces them, and from global names to entries in the global symbol table;

2. *type-checked*—in the context of a `TypeChecker` to discover whether it has a type at all, or several possible ones (only expressions that have a unique unambiguous type are further processed);

3. *"sort-sanitized"*—in the context of a `Sanitizer` to discriminate between those local variables that are of primitive Java types (`int` or `double`) or of `Object` type (this is necessary because the set-up means to use unboxed values of primitive types for efficiency reasons); this second "sanitization" phase is also used to compute offsets for local names (*i.e.*, so-called *de Bruijn indices*) for each type sort;

4. *compiled*—in the context of a `Compiler` to generate the sequence of instructions whose execution in an appropriate runtime environment will evaluate the expression;

5. *executed*—in the context of a `Runtime` to execute its sequence of instructions.

3.2.1 Sanitizer

A *sanitizer* is an object that "cleans up"—so to speak—an expression of its remaining ambiguities as it is being processed. There are two kinds of ambiguities that must be "sanitized:"

- after parsing, it must be determined which identifiers are the names of *local* variables *vs.* those of *global* variables;
- after type-checking, it must be determined the runtime sort of every abstraction parameter and use this to compute the local variable environment offsets of each local variable.[1]

Thus a sanitizer is a discriminator of names and sorts.[2]

3.2.2 Typechecker

The type checker is in fact a type inference machine that synthesizes missing type information by type unification. It may be (and often is) used as a type-checking automaton when types are (partially) present.

Each expression must specify its own `typeCheck(TypeChecker)` method that encodes its formal typing rule.

3.2.3 Compiler

This is the class defining a compiler object. Such an object serves as the common compilation context shared by an `Expression` and the subexpressions comprising it. Each type of expression representing a syntactic construct of the kernel language defines a `compile(Compiler)` method that specifies the way the construct is to be compiled in the context of a given compiler. Such a compiler object consists of attributes and methods for generating straightline code which consists of the sequence of instructions corresponding to a top-level expression and its subexpressions.

Upon completion of the compilation of a top-level expression, a resulting code array is extracted from the sequence of instructions, which may then be executed in the context of a `Runtime` object, or, in the case of a `Definition`, be saved in the code array in the `Definition`'s `codeEntry()` field—a `DefinedEntry` object, which encapsulates its code entry point, which in turn may then be used to access the defined symbol's code for execution).

[1] These offsets are the so-called *de Bruijn* indices of λ-calculus [4]. Or rather, their sorted version.

[2] It has occurred to this author that his choice of the word "sanitizer" is perhaps a tad of a misnomer—"discriminator" may be a better choice. This also goes for the `ilog.language.design.kernel.Sanitizer` class' method names (*i.e.*, `discriminateNames` and `discriminateSorts` rather than `sanitizeNames` and `sanitizeSorts`).

Each expression construct of the kernel must therefore specify a compiling rule. Such a rule expresses how the abstract syntax construct maps into a straightline code sequence.

3.3 Description of kernel expressions

The class `Expression` is the mother of all expressions in the kernel language. It specifies the prototypes of the methods that must be implemented by all expression subclasses. The subclasses of `Expression` are:

- `Constant`: constant (void, boolean, integer, real number, object);[3]
- `Abstraction`: functional abstraction (*à la* λ-calculus);[4]
- `Application`: functional application;
- `Local`: local name;
- `Parameter`: a function's formal parameter (really a pseudo-expression as it is not fully processed as a real expression and is used as a shared type information repository for all occurrences in a function's body of the variable it stands for);
- `Global`: global name;
- `Dummy`: temporary place holder in lieu of a name prior to being discriminated into a local or global one.
- `Definition`: definition of a global name with an expression defining it in a global store;
- `IfThenElse`: conditional;
- `AndOr`: non-strict boolean conjunction and disjunction;
- `Sequence`: sequence of expressions (presumably with side-effects);
- `Let`: lexical scoping construct;
- `Loop`: conditional iteration construct;
- `ExitWithValue`: non-local function exit;
- `Assignment`: construct to set the value of a `local` or a `global` variable;
- `NewArray`: construct to create a new (multidimensional) array;
- `ArraySlot`: construct to access the element of an array;
- `ArraySlotUpdate`: construct to update the element of an array;

[3] Section 3.3.1.
[4] Section 3.3.2.

- `Tuple`: construct to create a new position-indexed tuple;
- `NamedTuple`: construct to create a new name-indexed tuple;
- `TupleProjection`: construct to access the component of a tuple;
- `TupleUpdate`: construct to update the component of a tuple;
- `NewObject`: construct to create a new object;
- `DottedNotation`: construct to emulate the traditional object-oriented "dot" dereference notation;
- `FieldUpdate`: construct to update the value of an object's field;
- `ArrayExtension`: construct denoting a literal array;
- `ArrayInitializer`: construct denoting a syntactic convenience for specifying initialization of an array from an extension;
- `Homomorphism`: construct denoting a monoid homomorphism;
- `Comprehension`: construct denoting a monoid comprehension;

In this section, we are going to give a detailed description of each kernel construct. The description of an expression will have the following items:

- ABSTRACT SYNTAX
- OPERATIONAL SEMANTICS
- TYPING RULE
- COMPILING RULE

ABSTRACT SYNTAX

This describes the abstract syntax form of the kernel expression. A kernel expression will be written in $blue$.

OPERATIONAL SEMANTICS

This describes informally the meaning of the expression. The notation $[\![e]\!]$, where e is an abstract syntax expression, denotes the (mathematical) semantic *denotation* of e. The notation $[\![T]\!]$, where T is a type, denotes the (mathematical) semantic *denotation* of T—namely, $[\![T]\!]$ is the set of all abstract denotations $[\![e]\!]$'s such that kernel expression e has type T.

TYPING RULE

This describes formally the logical rules for typing the kernel expression. A type will be written in red.

A *typing judgment* is a formula of the form $\Gamma \vdash e : T$, and is read as: "*under typing context* Γ, *expression* e *has type* T."

In its simplest form, a *typing context* Γ is a function mapping the parameters of λ-abstractions to their types. In the formal presentation of an expression's typing rule, the context keeps the type binding under which the typing derivation has progressed up to applying the rule in which it occurs.

The notation $\Gamma[x : T]$ denotes the context defined from Γ as follows:

$$\Gamma[x : T](y) \stackrel{\text{def}}{=} \begin{cases} T & \text{if } y = x; \\ \Gamma(x) & \text{otherwise.} \end{cases} \tag{3.1}$$

A *typing rule* is a formula of the form:

$$\frac{J_1, \ldots, J_n}{J} \tag{3.2}$$

where J and the J_i's, $i = 0, \ldots, n,\ n \geq 0$, are typing judgments. This "fraction" notation expresses essentially an implication: when all the formulae of the rule's *premises* (the J_i's in the fraction's "numerator") hold, then the formula in the rule's *conclusion* (the fraction's "denominator") holds too. When $n = 0$, the rule has no premise—*i.e.*, the premise is tautologically *true* (*e.g.*, $0 = 0$)—the rule is called an *axiom* and is written with an empty "numerator."

A *conditional* typing rule is a typing rule of the form:

$$\frac{J_1, \ldots, J_n}{J} \ \texttt{if}\ \ c(J_1, \ldots, J_n) \tag{3.3}$$

where c is a boolean metacondition involving the rule's judgments.

A typing rule (or axiom), whether or not in conditional form, is usually read backwards (*i.e.*, upwards) from the rule's *conclusion* (the bottom part, or "denominator") to the rule's *premises* (the top part, or "numerator"). Namely, the rule of the form:

$$\frac{\Gamma_1 \vdash e_1 : T_1,\ \ldots,\ \Gamma_n \vdash e_n : T_n}{\Gamma \vdash e : T} \tag{3.4}$$

is read thus:

> *"The expression* e *has type* T *under typing context* Γ ***if*** *the expression* e_1 *has type* T_1 *under typing context* Γ_1, ***and*** $\ldots$, *the expression* e_n *has type* T_n *under typing context* Γ_n*."*

For example,

$$\frac{\Gamma \vdash c : \mathfrak{Boolean},\ \Gamma \vdash e_1 : T,\ \Gamma \vdash e_2 : T}{\Gamma \vdash \mathfrak{if}\ c\ \mathfrak{then}\ e_1\ \mathfrak{else}\ e_2 : T}$$

is read thus:

> *"The expression* $\mathfrak{if}\ c\ \mathfrak{then}\ e_1\ \mathfrak{else}\ e_2$ *has type* T *under typing context* Γ ***if*** *the expression* c *has type* $\mathfrak{Boolean}$ *under typing context* Γ ***and*** *if both expressions* e_1 *and* e_2 *have the same type* T *under the same typing context* Γ*."*

With judgments spelled-out, a conditional typing rule (3.3) looks like:

$$\frac{\Gamma_1 \vdash e_1 : T_1,\ \ldots,\ \Gamma_n \vdash e_n : T_n}{\Gamma \vdash e : T}\ \textbf{\textit{if}}\ c(\Gamma, \Gamma_1, \ldots, \Gamma_n, e, e_1, \ldots, e_n, T, T_1, \ldots, T_n) \quad (3.5)$$

where $c(\Gamma, \Gamma_1, \ldots, \Gamma_n, e, e_1, \ldots, e_n, T, T_1, \ldots, T_n)$ is a boolean meta-condition involving the contexts, expressions, and types. Such a rule is read thus:

> *"****if*** *the meta-condition holds,* ***then*** *the expression* e *has type* T *under typing context* Γ ***if*** *the expression* e_1 *has type* T_1 *under typing context* Γ_1*,* ***and*** *...., the expression* e_n *has type* T_n *under typing context* Γ_n*."*

An example of a conditional rule is that of abstractions that must take into account whether or not the abstraction is *exitable—i.e.*, it may be exited non-locally:[5]

$$\frac{\Gamma[x_1 : T_1] \cdots [x_n : T_n] \vdash e : T}{\Gamma \vdash \mathfrak{function}\ x_1, \ldots, x_n \cdot e : T_1, \ldots, T_n \to T}\ \textbf{\textit{if}}\ \mathfrak{function}\ x_1, \ldots, x_n \cdot e \text{ is not exitable.}$$

Similarlly, a *typing axiom*:

$$\frac{}{\Gamma \vdash e : T} \quad (3.6)$$

is read as, *"The expression* e *has type* T *under typing context* Γ*."* and a *conditional typing axiom* is a typing axiom of the form:

$$\frac{}{\Gamma \vdash e : T}\ \textbf{\textit{if}}\ c(\Gamma, e, T) \quad (3.7)$$

where $c(\Gamma, e, T)$ is a boolean meta-condition on typing context Γ, expression e, and type T and is read as, *"****if*** *the meta-condition* $c(\Gamma, e, T)$ *holds* ***then*** *the expression* e *has type* T *under typing context* Γ*."* We shall see examples of typing axioms in Sections 3.3.1 and 3.3.5.

[5] See Sections 3.3.2 and 3.3.14.

COMPILING RULE

This describes the way the expression's components are mapped into a straightline sequence of instructions. An instruction (or generally any instruction sequence) will be written in MAGENTA. Any meta-information annotation used in code instructions or instruction sequences will be written in ***green***.

The compiling rule for expression e is given as a function $\mathfrak{compile}[\![\_]\!]$ of the form:

$$\begin{aligned}\mathfrak{compile}[\![e]\!] = \quad & \texttt{INSTRUCTION}_1 \\ & \vdots \\ & \texttt{INSTRUCTION}_n\end{aligned} \tag{3.8}$$

3.3.1 Constant

Constants represents the built-in primitive (unconstructed) data elements of the kernel language.

ABSTRACT SYNTAX

A *Constant* expression is an atomic literal. Objects of class `Constant` denote literal constants: the integers (*e.g.*, -1, 0, 1, *etc.*), the real numbers (*e.g.*, -1.23, ..., 0.0, ..., 1.23, *etc.*), the characters (*e.g.*, $'a'$, $'b'$, $'@'$, $'\#'$, *etc.*), and the constants $\mathfrak{void}$, $\mathfrak{true}$, and $\mathfrak{false}$. The constant $\mathfrak{void}$ is of type $\mathfrak{Void}$, such that:

$$[\![\mathfrak{Void}]\!] \stackrel{\text{def}}{=} \{[\![\mathfrak{void}]\!]\}$$

and the constants $\mathfrak{true}$ and $\mathfrak{false}$ of type $\mathfrak{Boolean}$, such that:

$$[\![\mathfrak{Boolean}]\!] \stackrel{\text{def}}{=} \{[\![\mathfrak{false}]\!], [\![\mathfrak{true}]\!]\}.$$

Other built-in types are:

$$\begin{aligned}[\![\mathfrak{Int}]\!] &\stackrel{\text{def}}{=} \mathbb{Z} = \{\ldots, [\![-1]\!], [\![0]\!], [\![1]\!], \ldots\} \\ [\![\mathfrak{Real}]\!] &\stackrel{\text{def}}{=} \mathbb{R} = \{\ldots, [\![-1.23]\!], \ldots, [\![0.0]\!], \ldots, [\![1.23]\!], \ldots\} \\ [\![\mathfrak{Char}]\!] &\stackrel{\text{def}}{=} \text{set of all Unicode characters} \\ [\![\mathfrak{String}]\!] &\stackrel{\text{def}}{=} \text{set of all finite strings of Unicode characters.}\end{aligned}$$

Thus, the `Constant` expression class is further subclassed into: `Int`, `Real`, `Char`, `NewObject`, and `BuiltinObjectConstant`, whose instances denote, respectively: integers, floating-point numbers, characters, new objects, and built-in object constants (*e.g.*, strings).

TYPING RULE

The typing rules for each kind of constant are:

$$\texttt{[void]}\ \frac{}{\Gamma \vdash \texttt{void} : \mathfrak{Void}}$$

$$\texttt{[true]}\ \frac{}{\Gamma \vdash \texttt{true} : \mathfrak{Boolean}}$$

$$\texttt{[false]}\ \frac{}{\Gamma \vdash \texttt{false} : \mathfrak{Boolean}}$$

$$\texttt{[int]}\ \frac{}{\Gamma \vdash n : \mathfrak{Int}}\quad \textit{if } n \text{ is an integer} \tag{3.9}$$

$$\texttt{[real]}\ \frac{}{\Gamma \vdash n : \mathfrak{Real}}\quad \textit{if } n \text{ is a floating-point number}$$

$$\texttt{[char]}\ \frac{}{\Gamma \vdash c : \mathfrak{Char}}\quad \textit{if } c \text{ is a character}$$

$$\texttt{[string]}\ \frac{}{\Gamma \vdash s : \mathfrak{String}}\quad \textit{if } s \text{ is a string}$$

We postpone for now the typing of object constants until we understand object types.

3.3.2 Abstraction

ABSTRACT SYNTAX

This is the standard λ-calculus functional abstraction, possibly with multiple parameters. Rather than using the conventional λ notation, we write an abstraction as:

$$\mathfrak{function}\ x_1, \ldots, x_n \cdot e \tag{3.10}$$

where the x_i's are *abstraction parameters*—identifiers denoting variables local to the expression e, the abstraction's *body*.

TYPING RULE

There are two cases to consider depending on whether the abstraction is or not *exitable*. An exitable abstraction is one that corresponds to a real source language's function from which a user may exit non-locally.[6] Other (non-exitable) abstractions are those that are implicitly generated by syntactic desugaring of surface syntax—*e.g.*, see Sections 3.3.12 and 3.3.29. It is the responsibility of the parser to identify the two kinds of abstractions and mark as exitable all and only those abstractions that should be.

$$\frac{\Gamma[x_1 : T_1] \cdots [x_n : T_n] \vdash e : T}{\Gamma \vdash \mathfrak{function}\, x_1, \ldots, x_n \cdot e : T_1, \ldots, T_n \to T} \quad \textit{if} \quad \mathfrak{function}\, x_1, \ldots, x_n \cdot e \text{ is not exitable} \tag{3.11}$$

If the abstraction is exitable however, we must record it in the typing context. Namely,let $a = \mathfrak{function}\, x_1, \ldots, x_n \cdot e$; then,

$$\frac{\Gamma_{\aleph \leftarrow a}[x_1 : T_1] \cdots [x_n : T_n] \vdash e : T}{\Gamma \vdash a : T_1, \ldots, T_n \to T} \quad \textit{if} \quad a \text{ is exitable} \tag{3.12}$$

where $\Gamma_{\aleph \leftarrow a}$ is the same context as Γ except that $\aleph_{\Gamma_{\aleph \leftarrow a}} \stackrel{\text{def}}{=} a$.

3.3.3 Application

ABSTRACT SYNTAX

$$f(e_1, \ldots, e_n) \tag{3.13}$$

TYPING RULE

$$\frac{\Gamma \vdash e_1 : T_1, \cdots, \Gamma \vdash e_n : T_n, \quad \Gamma \vdash f : T_1, \ldots, T_n \to T}{\Gamma \vdash f(e_1, \ldots, e_n) : T} \tag{3.14}$$

[6] See $\mathfrak{exit\ with}\ v$ in Section 3.3.14, on Page 17.

3.3.4 Local

3.3.5 Parameter

3.3.6 Global

3.3.7 Dummy

3.3.8 Definition

3.3.9 IfThenElse

ABSTRACT SYNTAX

$$\mathfrak{if}\ c\ \mathfrak{then}\ e_1\ \mathfrak{else}\ e_2$$

OPERATIONAL SEMANTICS

TYPING RULE

$$\frac{\Gamma \vdash c : \mathfrak{Boolean},\ \Gamma \vdash e_1 : T,\ \Gamma \vdash e_2 : T}{\Gamma \vdash \mathfrak{if}\ c\ \mathfrak{then}\ e_1\ \mathfrak{else}\ e_2 : T} \tag{3.15}$$

COMPILING RULE

$$\begin{aligned} \mathfrak{compile}[\![\mathfrak{if}\ c\ \mathfrak{then}\ e_1\ \mathfrak{else}\ e_2]\!] = \quad & \mathfrak{compile}[\![c]\!] \\ & \textsc{Jump\_On\_False}\ \mathit{jof} \\ & \mathfrak{compile}[\![e_1]\!] \\ & \textsc{Jump}\ \mathit{jmp} \\ \mathit{jof}: \quad & \mathfrak{compile}[\![e_2]\!] \\ \mathit{jmp}: \quad & \ldots \end{aligned} \tag{3.16}$$

3.3.10 AndOr

ABSTRACT SYNTAX

e_1 and/or e_2

TYPING RULE

$$\frac{\Gamma \vdash e_1 : \mathfrak{Boolean},\ \Gamma \vdash e_2 : \mathfrak{Boolean}}{\Gamma \vdash e_1 \text{ and/or } e_2 : \mathfrak{Boolean}} \tag{3.17}$$

And

COMPILING RULE

$$\begin{aligned}
\mathfrak{compile}[\![e_1 \text{ and } e_2]\!] = \quad & \mathfrak{compile}[\![e_1]\!] \\
& \text{JUMP_ON_FALSE } \mathit{jof} \\
& \mathfrak{compile}[\![e_2]\!] \\
& \text{JUMP_ON_TRUE } \mathit{jot} \\
\mathit{jof}: \quad & \text{PUSH_FALSE} \\
& \text{JUMP } \mathit{jmp} \\
\mathit{jot}: \quad & \text{PUSH_TRUE} \\
\mathit{jmp}: \quad & \ldots
\end{aligned} \tag{3.18}$$

Or

COMPILING RULE

$$\begin{aligned}
\mathfrak{compile}[\![e_1 \text{ or } e_2]\!] = \quad & \mathfrak{compile}[\![e_1]\!] \\
& \text{JUMP_ON_TRUE } \mathit{jot} \\
& \mathfrak{compile}[\![e_2]\!] \\
& \text{JUMP_ON_FALSE } \mathit{jof} \\
\mathit{jot}: \quad & \text{PUSH_TRUE} \\
& \text{JUMP } \mathit{jmp} \\
\mathit{jof}: \quad & \text{PUSH_FALSE} \\
\mathit{jmp}: \quad & \ldots
\end{aligned} \tag{3.19}$$

3.3.11 Sequence

ABSTRACT SYNTAX

$$\{\ e_1;\ \ldots;e_n\ \}$$

TYPING RULE

$$\frac{\Gamma\ \vdash\ e_1\ :\ T_1,\ \ \ldots,\ \ \Gamma\ \vdash\ e_n\ :\ T_n}{\Gamma\ \vdash\ \{\ e_1;\ \ldots;e_n\ \}\ :\ T_n} \tag{3.20}$$

COMPILING RULE

$$\mathtt{compile}[\![\{\ e_1;\ \ldots;e_n\ \}]\!] = \begin{array}{l}\mathtt{compile}[\![e_1]\!]\\ \textsc{Pop}\_sort(e_1)\\ \vdots\\ \mathtt{compile}[\![e_n]\!]\end{array} \tag{3.21}$$

3.3.12 Let

3.3.13 Loop

ABSTRACT SYNTAX

$$\mathfrak{while}\ c\ \mathfrak{do}\ e \tag{3.22}$$

where c and e are expressions.

OPERATIONAL SEMANTICS

TYPING RULE

$$\frac{\Gamma\ \vdash\ c\ :\ \mathfrak{Boolean},\ \ \Gamma\ \vdash\ e\ :\ T}{\Gamma\ \vdash\ \mathfrak{while}\ c\ \mathfrak{do}\ e\ :\ \mathfrak{Void}} \tag{3.23}$$

COMPILING RULE

$$\begin{array}{rll} \mathit{compile}[\![\mathfrak{while}\ c\ \mathfrak{do}\ e]\!] = & \mathit{loop}: & \mathit{compile}[\![c]\!] \\ & & \textsc{Jump\_On\_False}\ \mathit{jof} \\ & & \mathit{compile}[\![e]\!] \\ & & \textsc{Jump}\ \mathit{loop} \\ & \mathit{jof}: & \textsc{Push\_Void} \end{array} \tag{3.24}$$

3.3.14 ExitWithValue

ABSTRACT SYNTAX

$$\mathfrak{exit}\ \mathfrak{with}\ v \tag{3.25}$$

where v is an expression.

OPERATIONAL SEMANTICS

Normally, exiting from an abstraction is done simply by "falling off" (one of) the tip(s) of the expression tree of the abstraction's body. This operation is captured by the simple operational semantics of each of the three RETURN instructions. Namely, when executing a RETURN instruction, the runtime performs the following three-step procedure:

1. it pops the result from its result stack;[7]
2. it restores the latest saved runtime state (popped off the saved-state stack);
3. it pushes the result popped in Step 1 onto the restored state's own result stack.

However, it is also often desirable, under certain circumstances, that computation *not* be let to proceed further at its current level of nesting of exitable abstractions. Then, computation may be allowed to return right away from this current nesting (*i.e.*, as if having fallen off this level of exitable abstraction) when the conditions for this to happen are met. Exiting an abstraction thus must also return a specific value that may be a function of the context. This is what the kernel construction $\mathfrak{exit}\ \mathfrak{with}\ v$ expresses. This kernel construction is provided in order to specify that the current local computation should terminate without further ado, and exit with the value denoted by the specified expression.

[7] Where *stack* here means "stack of *appropriate* runtime sort;" appropppriate, that is, as per the instruction's sort—*viz.*, INT, REAL, or runtime OBJECT.

TYPING RULE

Now, there are several notions in the above paragraphs that need some clarification. For example, what an "*exitable*" abstraction is, and why worry about a dedicated construct in the kernel language for such a notion if it does nothing more than what is done by a RETURN instruction.

First of all, from its very name $\mathfrak{exit\ with}\ v$ assumes that computation has *entered* that from which it must *exit*. This is an *exitable* abstraction; that is, the latest λ-abstraction having the property of being *exitable*. Not all abstractions are exitable. For example, any abstraction that is generated as part of the target of some other kernel expression's syntacting sugar (*e.g.*, $\mathfrak{let}\ x_1 = e_1; \ldots; x_n = e_n;\ \mathfrak{in}\ e$ or $\langle \oplus, \mathbb{1}_\oplus \rangle \{e \mid x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n\}$, and more generally any construct that hide implicit abstractions within), will *not* be deemed exitable.

Secondly, exiting with a value v means that the type T of v must be congruent with what the return type of the abstraction being exited is. In other words:

$$\frac{\Gamma \vdash \aleph_\Gamma : T' \to T,\ \ \Gamma \vdash v : T}{\Gamma \vdash \mathfrak{exit\ with}\ v : T} \tag{3.26}$$

where $\aleph_\Gamma$ denotes the latest *exitable* abstraction in the context Γ.

The above scheme indicates the following necessities:

1. The typing rules for an abstraction deemed exitable must record in its typing context Γ the latest exitable abstraction, if any such exists; (if none does, a static semantics error is triggered to indicate that it is impossible to exit from anywhere before first entering somewhere).[8]
2. Congruently, the APPLY instruction of an exitable closure must take care of chaining this exitable closure before it pushes a new state for it in the saved state stack of the runtime system with the last saved exitable closure, and mark the saved state as being exitable; (dually, this exitable state stack must also be popped upon "falling off"—*i.e.*, normally exiting—an exitable closure. That is, whenever an exitable state is restored).
3. New NL_RETURN instructions (for each runtime sort) must be defined like their corresponding RETURN instructions except that the runtime state to restore is the one popped out of the exitable state stack.

COMPILING RULE

$$\begin{array}{ll} \mathfrak{compile}[\![\mathfrak{exit\ with}\ v]\!] = & \mathfrak{compile}[\![v]\!] \\ & \mathsf{NL\_RETURN\_}\mathit{sort}(v) \end{array} \tag{3.27}$$

[8] See Typing Rule 3.12 on Page 13.

3.3.15 Assignment

3.3.16 NewArray

3.3.17 ArraySlot

3.3.18 ArraySlotUpdate

3.3.19 ArrayExtension

3.3.20 ArrayInitializer

3.3.21 Tuple

3.3.22 NamedTuple

3.3.23 TupleProjection

3.3.24 TupleUpdate

3.3.25 NewObject

3.3.26 DottedNotation

This class represents a syntactic construct that is often used, albeit with different, though related, interpretations. A dotted notation is an expression (most often an application, but it could be any composition—of *what/with what* to be determined according to partial type analysis).

Thus, this class can be used to represent a particular kind of functional applications *à la* object-oriented programming; or (equivalently), arrow composition in Category Theory. More precisely, it represents the application (resp., composition) of an `Expression` to (resp., with) another `Expression`, those `Expression`s being determined according to the type of the expression on the *left* of the ”dot”.

Thus, a dotted notation is interpreted as follows.

A `DottedNotation` object is a wrapper of an expression—most often, of an application, but more generally, of any composition. It is a binary expression of the form $e_1.e_2$, where e_1 and e_2 are expressions. This is interpreted depending on the type of e_1 as follows:

- if e_1's type is a class `C`, then this is interpreted as:

  $\mathfrak{mangle}(e_1)\mathfrak{args}(e_1)$

  where $\mathfrak{mangle}(()$o:C,e) is some "mangling" of the name of the member of the object expression o of class type C being so-referred, and $\mathfrak{args}(e)$ is either the empty string or the args of the member expression e.

  For example, consider:

  ```
  counter.set(1);
  ```

  when `counter : Counter` is an object of class `Counter`, with method (say) `Counter_set : (Counter, int) -> int`. Then, for example, a default "mangling" $\mathfrak{mangle}(C, member)$ may simply concatenate the names of the class and the member separated with an underscore character ('_'), followed by '`(`', the object, and '`)`'; in other words: $\mathfrak{mangle}(counter : Counter, set(1))$ = "`Counter_set(counter)`" and $\mathfrak{args}(\texttt{set(1)} = \text{"(1)"})$. Therefore,

  ```
  counter.set(1) ====> Counter_set(counter)(1);
  ```

  that is:

  ```
  counter.set(1) ====> Counter_set(counter,1);
  ```

- if e_1's type is a tuple type $\langle T_1, \ldots, T_n \rangle$ or $\langle l_1 : T_1, \ldots, l_n : T_n \rangle$, then this is interpreted as:

  $\mathfrak{project}_{e_2}(e_1)$

  where $\mathfrak{project}_e$ is a tuple *projection* of type $\langle T_1, \ldots, T_n \rangle \rightarrow T_k$ and $\texttt{e} = \texttt{k}, k = 1, \ldots, n$, or of type $\langle l_1 : T_1, \ldots, l_n : T_n \rangle \rightarrow T_k$ and $\texttt{e} = \texttt{l}_\texttt{k}, k = 1, \ldots, n$. This is interpreted as:

  $$TupleProjection(e_1, e_2) : T_k.$$

  For example:

  $\langle \texttt{name} := \text{"a"}, \texttt{number} := \texttt{1} \rangle.\texttt{name} \rightsquigarrow \langle \texttt{name} := \text{"a"}, \texttt{number} := \texttt{1} \rangle @\texttt{name}$

  In other words,

$\mathfrak{proj}_{\texttt{n}}\texttt{ame} : \langle \texttt{name} : \texttt{string}, \texttt{number} : \texttt{int} \rangle \rightarrow \texttt{string}(\langle \texttt{name} := \text{"a"}, \texttt{number} := \texttt{1} \rangle) : \texttt{string}$

- Otherwise, the default is simply to interpret this as the application:

  $\texttt{member\_name}(\texttt{e}_2)(\texttt{e}_1)(\mathfrak{args}(\texttt{e}_2))$

  This default behavior can be overridden and customized through the methods: `setNoDefault()` and `setDefault(Expression)`.

3.3.27 FieldUpdate

3.3.28 Homomorphism

By *homomorphism* me mean specifically *monoid* homomorphism. For our purposes, a monoid is a set of data values or structures (*i.e.*, a data type) endowed with an associative binary operation and an identity element. Examples are:

Type	Operation	Identity
$\mathfrak{Int}$	$+_{\mathfrak{Int}}$	0
$\mathfrak{Int}$	$*_{\mathfrak{Int}}$	1
$\mathfrak{Int}$	$\mathfrak{max}_{\mathfrak{Int}}$	$-\infty_{\mathfrak{Int}}$
$\mathfrak{Int}$	$\mathfrak{min}_{\mathfrak{Int}}$	$+\infty_{\mathfrak{Int}}$
$\mathfrak{Real}$	$+_{\mathfrak{Real}}$	0.0
$\mathfrak{Real}$	$*_{\mathfrak{Real}}$	1.0
$\mathfrak{Real}$	$\mathfrak{max}_{\mathfrak{Real}}$	$-\infty_{\mathfrak{Real}}$
$\mathfrak{Real}$	$\mathfrak{min}_{\mathfrak{Real}}$	$+\infty_{\mathfrak{Real}}$
$\mathfrak{Boolean}$	$\mathfrak{or}_{\mathfrak{Boolean}}$	$\mathfrak{false}$
$\mathfrak{Boolean}$	$\mathfrak{and}_{\mathfrak{Boolean}}$	$\mathfrak{true}$
set data structures	set union	the empty set $\{\}$
list data structures	list concatenation	the empty list $[]$
...		

Monoid homomorphisms are quite useful for expressing a certain kind of iteration declaratively.

ABSTRACT SYNTAX

This is the class of objects denoting (monoid) homomorphisms. Such an expression means to iterate through a collection, applying a function to each element, accumulating the results along the way with an operation, and returning the end result. More precisely, it is the built-in version of the general computation scheme whose instance is the following *"hom"* functional, which may be formulated recursively, for the case of a list collection, as:

$$\begin{aligned} \mathbf{hom}^{\mathbb{1}_\oplus}_{\oplus}(f)[] &= \mathbb{1}_\oplus \\ \mathbf{hom}^{\mathbb{1}_\oplus}_{\oplus}(f)[H|T] &= f(H) \oplus \mathbf{hom}^{\mathbb{1}_\oplus}_{\oplus}(f)T \end{aligned} \tag{3.28}$$

Clearly, this scheme extends a function *f* to a homomorphism of monoids, from the monoid of lists to the monoid defined by $\langle \oplus, \mathbb{1}_{\oplus} \rangle$.

Thus, an object of this class denotes the result of applying such a homomorphic extension of a function (*f*) to an element of collection monoid (*i.e.*, a data structure such as a set, a list, or a bag), the image monoid being implicitly defined by the binary operation ($\oplus$)—also called the *accumulation* operation. It is made to work iteratively.

For technical reasons, we need to treat specially so-called *collection* homomorphisms; *i.e.*, those whose accumulation operation constructs a collection, such as a set. Although a collection homomorphism can conceptualy be expressed with the general scheme, the function applied to an element of the collection will return a collection (*i.e.*, a *free* monoid) element, and the result of the homomorphism is then the result of tallying the partial collections coming from applying the function to each element into a final "concatenation."

Other (non-collection) homomorphisms are called *primitive* homomorphisms. For those, the function applied to all elements of the collection will return a *computed* element that may be directly composed with the other results. Thus, the difference between the two kinds of (collection or primitive) homomorphisms will appear in the typing and the code generated (collection homomorphism requiring an extra loop for tallying partial results into the final collection). It is easy to make the distinction between the two kinds of homomorphisms thanks to the type of the accumulation operation (see below).

Therefore, a *collection homomorphism* expression constructing a collection of type $coll(T)$ consists of:

- the collection iterated over—of type $coll'(T')$;
- the iterated function applied to each element—of type $T' \rightarrow coll(T)$; and,
- the operation "adding" an element to a collection—of type $T, coll(T) \rightarrow coll(T)$.

T' *primitive homomorphism* computing a value of type T consists of:

- the collection iterated over—of type $coll'(T')$;
- the iterated function applied to each element—of type $T' \rightarrow T$; and,
- the monoid operation—of type $T, T \rightarrow T$.

Even though the scheme of computation for homomorphisms described above is correct, it is not often used, especially when the function already encapsulates the accumulation operation, as is always the case when the homomorphism comes from the desugaring of a *comprehension*—see below). Then, such a homomorphism will directly side-effect the collection structure specified as the identity element with a function of the form $\mathtt{function}\ x\ \cdot\ x \oplus \mathbb{1}_{\oplus}$ (*i.e.*, adding element x to the collection) and dispense altogether with the need to accumulate intermediate results. We shall call

those homomorphisms *in-place* homomorphisms. To distinguish them and enable the suprression of intermediate computations, a flag indicating that the homomorphism is to be computed in-place is provided. Both primitive and collection homomorphisms can be specified to be in-place. If nothing regarding in-place computation is specified for a homomorphism, the default behavior will depend on whether the homomorphism is collection (default is in-place), or primitive (default is *not* in-place). Methods to override the defaults are provided.

For an in-place homomorphism, the iterated function encapsulates the operation, which affects the identity element, which thus accumulates intermediate results and no further composition using the operation is needed. This is especially handy for collections that are often represented, for (space and time) efficiency reasons, by iteratable bulk structures constructed by allocating an empty structure that is filled in-place with elements using a built-in *"add"* method guaranteeing that the resulting data structure is canonical—*i.e.*, that it abides by the algebraic properties of its type of collection (*e.g.*, adding an element to a set will not create duplicates, *etc.*).

Although monoid homomorphisms are defined as expressions in the kernel, they are not meant to be represented directly in a surface syntax (although they could, but would lead to rather cumbersome and not very legible expressions). Rather, they are meant to be used for expressing higher-level expressions known as *monoid comprehensions*, which offer the advantage of the familar (set) comprehension notation used in mathematics, and can be translated into monoid homomorphisms to be type-checked and evaluated.

A monoid comprehension is an expression of the form:

$$\langle \oplus, \mathbb{1}_\oplus \rangle \{ e \mid q_1, \ldots, q_n \} \tag{3.29}$$

where $\langle \oplus, \mathbb{1}_\oplus \rangle$ define a monoid, e is an expression, and the q_i's are *qualifiers*. A qualifier is either an expression e or a pair $x \leftarrow e$, where x is a variable and e is an expression. The sequence of qualifiers may also be empty. Such a monoid comprehension is just syntactic sugar that can be expressed in terms of homomorphisms as follows:

$$\begin{aligned}
\langle \oplus, \mathbb{1}_\oplus \rangle \{ e \mid \} &\stackrel{\text{def}}{=} e \oplus \mathbb{1}_\oplus \\
\langle \oplus, \mathbb{1}_\oplus \rangle \{ e \mid x \leftarrow e', Q \} &\stackrel{\text{def}}{=} \mathfrak{hom}_\oplus^{\mathbb{1}_\oplus} [\lambda x. \langle \oplus, \mathbb{1}_\oplus \rangle \{ e \mid Q \}](e') \\
\langle \oplus, \mathbb{1}_\oplus \rangle \{ e \mid c, Q \} &\stackrel{\text{def}}{=} \mathfrak{if}\ c\ \mathfrak{then}\ \langle \oplus, \mathbb{1}_\oplus \rangle \{ e \mid Q \}\ \mathfrak{else}\ \mathbb{1}_\oplus
\end{aligned} \tag{3.30}$$

This is explained more formally in Section 3.3.29.

Comprehensions are also interesting as they may be subject to transformations leading to more efficient evaluation than their simple "nested loops" operational semantics (by using "unnesting" techniques and using relational operations as implementation instructions).

3.3.29 Comprehension

The concept of monoid homomorphism is useful for expressing a formal semantics of iteration over collections. However, it is not very convenient as a programming construct. A natural notation for such a construct that is both conspicuous and can be expressed in terms of monoid homomorphisms is a *monoid comprehension*. This notion generalizes the familiar notation used for writing a set in comprehension (as opposed to writing it in extension) using a pattern and a formula describing its elements (as oppposed to listing all its elements). For example, the set comprehension $\{\langle x, x^2\rangle \mid x \in \mathbb{N}, \exists n.x = 2n\}$ describes the set of pairs $\langle x, x^2\rangle$ (the *pattern*), verifying the formula $x \in \mathbb{N}, \exists n.x = 2n$ (the *qualifier*).

This notation can be extended to any (primitive or collection) monoid $\oplus$. The syntax of a monoid comprehension is an expression of the form $\oplus\{e \parallel Q\}$ where e is an expression called the *head* of the comprehension, and Q is called its qualifier and is a sequence $q_1, \ldots, q_n$, $n \geq 0$, where each q_i is either

- a *generator* of the form $x \leftarrow e$, where x is a variable and e is an expression; or,
- a *filter* ϕ which is a boolean condition.

In a monoid comprehension expression $\oplus\{e \parallel Q\}$, the monoid operation $\oplus$ is called the *accumulator*.

As for semantics, the meaning of a monoid comprehension is defined in terms of monoid homomorphisms.

DEFINITION 3.3.1 (MONOID COMPREHENSION) *The meaning of a monoid comprehension over a monoid $\oplus$ is defined inductively as follows:*

$$\oplus\{e \parallel \} \stackrel{def}{=} \begin{cases} \mathfrak{u}_\oplus(e) & \textit{if } \oplus \textit{ is a collection monoid} \\ e & \textit{if } \oplus \textit{ is a primitive monoid} \end{cases}$$

$$\oplus\{e \parallel x \leftarrow e', Q\} \stackrel{def}{=} \mathfrak{hom}^{\oplus}_{\odot}[\lambda x.\ \oplus\{e \parallel Q\}](e')$$

$$\oplus\{e \parallel c, Q\} \stackrel{def}{=} \mathfrak{if}\ c\ \mathfrak{then}\ \oplus\{e \parallel Q\}\ \mathfrak{else}\ \mathfrak{z}_\oplus$$

such that $e : \mathfrak{T}_\oplus$, $e' : \mathfrak{T}_\odot$, and $\odot$ is a collection monoid.

Note that although the input monoid $\oplus$ is explicit, each generator $x \leftarrow e'$ in the qualifier has an implicit collection monoid $\odot$ whose characteristics can be inferred with polymorphic typing rules.

Although Definition 3.3.1 can be effectively computed using nested loops (*i.e.*, using the iteration semantics (3.28)), such would be in general rather inefficient. Rather, an optimized implementation can be achieved by various syntactic transformation expressed as rewrite rules. Thus, the principal

benefit of using monoid comprehensions is to formulate efficient optimizations on a simple and uniform general syntax of expressions irrespective of specific monoids.

Thus, monoid comprehensions allow the formulation of "declarative iteration." Note the fact mentioned earlier that a homomorphism coming from the translation of a comprehension encapsulates the operation in its function. Thus, this is generally taken to advantage with operations that cause a side-effect on their second argument to enable an in-place homomorphism to dispense with unneeded intermediate computation.

Section 3.3.29 gives a detailed explanation of the syntactic desugaring of a pattern-directed high-level syntax of comprehensions into more basic kernel expressions.

Comprehension

This class represents a monoid comprehension whose actual form is interpreted as a construct involving the parts of the syntactic form of the comprehension. The syntax of a monoid comprehension is given by an expression of the form:

```
[op,id] { e | q1, ..., qn }
```

where `[op,id]` define a monoid, `e` is an expression, and the `qi`s are *qualifiers*. A qualifier is either a *boolean* expression or a pair `p <- e`, where `p` is a pattern (any expression) and `e` is an expression. The sequence of qualifiers may also be empty. Such a monoid comprehension is syntactic sugar that is in fact translated into a combination of homomorphisms and/or filtering tests, possibly wrapped inside a let factoring out some computation.

```
package ilog.langiage.design.kernel.Comprehension;

public class Comprehension extends ProtoExpression
{
  public static boolean OPAQUE_PARAMETERS = true;

  private Tables _tables;
  private RawInfo _raw;
  private Expression _construct;
  private Expression _operation;
  private Expression _identity;

  private Expression _enclosingScope;
```

Constructs an already translated comprehension as a *Let* construct. This is provided as a public constructor but should be used with care as it trusts that the specified arguments are correctly set up.

```
public Comprehension (AbstractList parameters, AbstractList values, Expression body)
  {
    _construct = new Let(parameters,values,body);
  }
```

Constructs a *raw* comprehension with the specifed arguments and assuming the default in-place mode for performing the monoid operation.

```
public Comprehension (Tables tables, Expression operation, Expression identity,
                      Expression expression, AbstractList patterns, AbstractList expressi
  {
    this(tables,operation,identity,expression,patterns,expressions,Homomorphism.DEFAULT_
  }
```

Constructs a *raw* comprehension with the specifed arguments. A comprehension is *raw* as long as it has not been translated into a meaningful expression. Translation will happen automatically as soon as the meaning expresssion is needed.

```
public Comprehension (Tables tables, Expression operation, Expression identity,
                      Expression expression, AbstractList patterns, AbstractList expressi
                      byte inPlace)
  {
    _tables = tables;

    _operation = operation;
    _identity = identity;

    if (patterns == null)
      patterns = expressions = new ArrayList(0);

    _raw = new RawInfo(new Dummy("$OP$").addTypes(operation).setExtent(operation),
                       new Dummy("$ID$").addTypes(identity).setExtent(identity),
                       expression,patterns,expressions,inPlace);
  }
```

Constructs a fully translated comprehension using the specified expression as its meaning expression.

```
private Comprehension (Expression construct)
  {
    _construct = construct;
  }
```

```
  public boolean _doLetWrapping = true;

  public final Comprehension setNoLetWrapping ()
    {
      _doLetWrapping = false;
      _raw.operation = _operation;
      _raw.identity = _identity;
      return this;
    }

  public final Tables tables ()
    {
      return _tables;
    }

  public final Expression operation ()
    {
      return _operation;
    }

  public final Expression identity ()
    {
      return _identity;
    }

  public final Expression copy ()
    {
      if (_raw == null)
        return new Comprehension(_construct.copy());

      ArrayList patterns = new ArrayList(_raw.patterns.size());
      for (int i=0; i<_raw.patterns.size(); i++)
        {
          Expression pattern = (Expression)_raw.patterns.get(i);
          if (pattern != null) patterns.add(pattern.copy());
        }

      ArrayList expressions = new ArrayList(_raw.expressions.size());
      for (int i=0; i<_raw.expressions.size(); i++)
        expressions.add(((Expression)_raw.expressions.get(i)).copy());

      return new Comprehension(_tables,_operation.copy(),_identity.copy(),
                               _raw.expression.copy(),patterns,expressions,
                               _raw.inPlace);
    }

  public final Expression typedCopy ()
    {
```

```
    if (_raw == null)
      return new Comprehension(_construct.typedCopy());

    ArrayList patterns = new ArrayList(_raw.patterns.size());
    for (int i=0; i<_raw.patterns.size(); i++)
      {
        Expression pattern = (Expression)_raw.patterns.get(i);
        if (pattern != null) patterns.add(pattern.typedCopy());
      }

    ArrayList expressions = new ArrayList(_raw.expressions.size());
    for (int i=0; i<_raw.expressions.size(); i++)
      expressions.add(((Expression)_raw.expressions.get(i)).typedCopy());

    return new Comprehension(_tables,_operation.typedCopy(),_identity.typedCopy(),
                             _raw.expression.typedCopy(),patterns,expressions,
                             _raw.inPlace).addTypes(this);
  }

public final int numberOfSubexpressions ()
  {
    if (_raw != null) _construct();
    return _construct.numberOfSubexpressions();
  }

public final Expression subexpression (int n) throws NoSuchSubexpressionException
  {
    if (_raw != null) _construct();
    return _construct.subexpression(n);
  }

public final Expression setSubexpression (int n, Expression expression) throws NoSuchSube
  {
    if (_raw != null) _construct();
    return _construct.setSubexpression(n,expression);
  }

public final Expression sanitizeNames (ParameterStack parameters, ClassTypeHandle handle)
  {
    if (_raw != null) _construct();
    _construct = _construct.sanitizeNames(parameters,handle);
    return this;
  }

public final void sanitizeSorts (Enclosure enclosure)
  {
    _construct.sanitizeSorts(enclosure);
  }
```

Constructs this monoid comprehension by first desugaring its patterns into simple parameters, then normalizing its qualifiers by unnestings filters as far to the left as possible, and finally translating the transformed raw comprehension into its meaning expression.

```
private final void _construct () throws UndefinedEqualityException
  {
    desugarPatterns();
    unnestInnerFilters();
  }
```

Returns the comprehension obtained after applying the specified substitution to the subexpressions of this. If this comprehension is already translated, this simply amounts to setting the construct to the substituted construct. If this is a raw comprehension, care must be taken to proceed from left to right over the qualifiers and preventing generator variables to be substituted in expressions lying to their right (including the main expression of the comprehension).

```
public final Expression substitute (HashMap substitution)
  {
    if (_raw == null)
      {
        _construct = _construct.substitute(substitution);
        return this;
      }

    if (!substitution.isEmpty())
      {
        _operation = _operation.substitute(substitution);
        _identity = _identity.substitute(substitution);

        _substituteQualifiers(0,substitution);
      }

    return this;
  }
```

Proceeds through the raw qualifiers substituting expressions making sure that generator parameters are removed from the substitution before applying it to what lies to the right of the specified index (including the main expression of the comprehension).

```
private final void _substituteQualifiers (int index, HashMap substitution)
  {
    if (index == _raw.patterns.size())
```

```
      _raw.expression = _raw.expression.substitute(substitution);
    else
      {
        Expression pattern = (Expression)_raw.patterns.get(index);
        Expression expression = (Expression)_raw.expressions.get(index);

        // in all cases, apply the substitution to the qualifying expression
        _raw.expressions.set(index,expression.substitute(substitution));

        if (pattern == null)
          // this is a filter - simply proceed
          _substituteQualifiers(index+1,substitution);
        else
          // this is a generator - must check whether pattern is opaque parameter
          if (pattern instanceof Parameter || (pattern instanceof Dummy && OPAQUE_PARAME
            {
              // this is an opaque parameter - it is removed from the substitution
              // before proceeding further to the right, and reinstated afterwards
              String name = pattern instanceof Dummy ? ((Dummy)pattern).name()
                                                     : ((Parameter)pattern).name();
              Object value = substitution.remove(name);
              _substituteQualifiers(index+1,substitution);
              if (value != null) substitution.put(name,value);
            }
          else
            { // this is not a parameter - apply the substitution to the pattern and proc
              _raw.patterns.set(index,pattern.substitute(substitution));
              _substituteQualifiers(index+1,substitution);
            }
      }
  }
```

Sets the link to the enclosing scope of this comprehension to the specified expression, then visits all the qualifier expressions to link up their scope trees of nested comprehensions to this, and returns the number of such nested comprehensions.

```
final int linkScopeTree (Expression ancestor)
  {
    if (_scopeTreeIsLinked)
      return _nestedComprehensionCount;

    _enclosingScope = ancestor;
    _nestedComprehensionCount = _raw.expression.linkScopeTree(this);

    for (int i=_raw.expressions.size(); i-->0;)
      _nestedComprehensionCount += ((Expression)_raw.expressions.get(i)).linkScopeTree(t
```

```
    _scopeTreeIsLinked = true;

    return 1 + _nestedComprehensionCount;
  }
```

Desugars the patterns of this comprehension into simple parameters, substituting expression in terms of these parameters inside the comprehension where appropriate. Then, this proceeds desugaring the patterns of nested comprehensions if any. It is important for this method to proceed top down so that the patterns of potential inner comprehensions may be affected by the desugaring of outer ones.

```
final void desugarPatterns () throws UndefinedEqualityException
  {
    if (_raw == null || _raw.isDesugared)
      return;

    _desugarPatterns();

    if (_nestedComprehensionCount > 0)
      {
        for (int i=0; i<_raw.patterns.size(); i++)
          {
            Expression pattern = (Expression)_raw.patterns.get(i);
            if (pattern != null) pattern.desugarPatterns();
            ((Expression)_raw.expressions.get(i)).desugarPatterns();
          }

        _raw.expression.desugarPatterns();
      }
  }
```

Converts the patterns into simple parameters and substitutes free occurrences of the formal names from the patterns by what is appropriate in terms of the new parameters inside the raw expression (and any other pertinent expression in raw expressions—*i.e.*, those to the right of a pattern generator). While desugaring, new filters may be generated along the way upon repeated occurrences of formal names or the presence of interpretable expressions in the patterns. These are simply appended to the raw list of expressions. Because of this, we need to append as many `null`s to the list of patterns in order to maintain the two lists at equal lengths.

```
private final void _desugarPatterns () throws UndefinedEqualityException
  {
    HashMap substitution = _initialSubstitution();

    int size = _raw.patterns.size();
```

```
    for (int i=0; i<size; i++)
      _raw.patterns.set(i,_desugarPattern(i,(Expression)_raw.patterns.get(i),substitutio

    for (int i=_raw.expressions.size()-size; i-->0;) _raw.patterns.add(null);

    _substituteDesugaring(substitution);

    _raw.isDesugared = true;
  }
```

Returns a substitution initialized with the local parameters of the enclosing scopes of this comprehension if any.

```
private final HashMap _initialSubstitution ()
  {
    HashMap substitution = new HashMap();

    for (Expression e = _enclosingScope; e != null; e = e.enclosingScope())
      if (e instanceof Scope)
        {
          Scope s = (Scope)e;
          for (int i = s.arity(); i-->0;)
            {
              String name = s.parameter(i).name();
              if (substitution.get(name) == null)
                substitution.put(name,
                                 new IndexedExpression(new Dummy(s.parameter(i))));
            }
        }

    return substitution;
  }
```

Transforms the specified pattern into a parameter and records in the specified substitution any appropriate expression in terms of this parameter for corresponding occurrences of the pattern components in the qualifying expressions at index higher than the specified index.

In `OPAQUE_PARAMETERS` mode (the default), an outer pattern consisting of just an identifier is always considered new and creates an opaque scope for its free occurrences in the qualifier expressions lying on its right as well as for the main expression of the comprehension. If on the other hand `OPAQUE_PARAMETERS` is `false`, such an identifier is deemed sensitive to its namesakes in the substitution and global scalar (*i.e.*, non-functional) definitions. Then, it will be considered a repeated or interpreted occurrence, whichever the case may be. It returns the parameter desugaring the specified pattern.

```
private final Parameter _desugarPattern (int index, Expression pattern, HashMap substitut
  throws UndefinedEqualityException
  {
    if (pattern == null || pattern instanceof Parameter)
      return (Parameter)pattern;

    Parameter parameter = null;
    Dummy variable = null;

    if (pattern instanceof Dummy)
      { // the pattern is an identifier
        variable = (Dummy)pattern;
        parameter = new Parameter(variable);

        if (!OPAQUE_PARAMETERS)
          {
            IndexedExpression value = (IndexedExpression)substitution.get(variable.name(

            if (value == null)
              { // this is the first occurrence - record only if not a global scalar
                if (!_tables.isDefinedScalar(variable.name()))
                  substitution.put(variable.name(),new IndexedExpression(index,variable)
              }
            else
              { // this is a repeated occurrence - generate an equality filter
                variable = new Dummy(parameter = new Parameter(value.expression.typeRef(
                _raw.expressions.add(new Application(_tables.equality(),
                                                    variable,
                                                    value.expression.typedCopy()));
              }
          }

        return parameter;
      }

    parameter = new Parameter(pattern.typeRef());
    variable = new Dummy(parameter.name());

    if (pattern instanceof Tuple)
      // the pattern is a tuple - proceed with desugaring it
      _desugarTuplePattern(index,(Tuple)pattern,variable,substitution);
    else
      // the pattern is an interpreted expression - generate an equality filter
      _raw.expressions.add(new Application(_tables.equality(),variable,pattern));

    return parameter;
  }
```

Desugars the specified `tuple` pattern given that the expression in which it is immediately nested is the expression specified as `father`.

```
  private final void _desugarTuplePattern (int index, Tuple tuple, Expression father,
                                           HashMap substitution)
    throws UndefinedEqualityException
    {
      if (tuple instanceof NamedTuple)
        { // treat named tuples specially
          _desugarNamedTuplePattern(index,(NamedTuple)tuple,father,substitution);
          return;
        }

      int dimension = tuple.dimension();
      for (int i=0; i<dimension; i++)
        // desugar each tuple component using the appropriate tuple projection as father
        _desugarTupleComponent(index,
                               tuple.component(i),
                               new TupleProjection(father,new Int(i+1)),
                               substitution);
    }
```

Desugars the specified named `tuple` pattern given that the expression in which it is immediately nested is the expression specified as `father`.

```
  private final void _desugarNamedTuplePattern (int index, NamedTuple tuple, Expression fa
                                                HashMap substitution)
    throws UndefinedEqualityException
    {
      TupleFieldName[] fields = tuple.fields();
      int dimension = fields.length;
      for (int i=0; i<dimension; i++)
        // desugar each tuple component using the appropriate tuple projection as father
        _desugarTupleComponent(index,
                               tuple.component(i),
                               new TupleProjection(father,new StringConstant(fields[i].nam
                               substitution);
    }
```

Desugars the specified tuple component corresponding to the specified tuple projection.

```
private final void _desugarTupleComponent (int index, Expression component,
                                           TupleProjection projection, HashMap substitutio
    throws UndefinedEqualityException
    {
```

```
    if (component instanceof Dummy)
      { // it is a leaf consisting of a name
        Dummy variable = (Dummy)component;
        IndexedExpression value = (IndexedExpression)substitution.get(variable.name());

        if (value == null && !_tables.isDefinedScalar(variable.name()))
          // record only if first occurrence and not a global scalar
          substitution.put(variable.name(),new IndexedExpression(index,projection));
        else
          // it is a repeated occurrence or a global scalar - generate an equality filter
          _raw.expressions.add(new Application(_tables.equality(),
                                               projection,
                                               variable.typedCopy()));

        return;
      }

    if (component instanceof Tuple)
      // it is a nested tuple pattern - desugar the nested pattern
      _desugarTuplePattern(index,(Tuple)component,projection,substitution);
    else
      // it is an interpreted expression - generate an equality filter
      _raw.expressions.add(new Application(_tables.equality(),projection,component));
  }
```

Applies the desugaring substitutions to each qualifier expression and the main expression, taking care of enabling only those substitutions at indices less than the index of the qualifier (and all of them for the main expression).

```
  private final void _substituteDesugaring (HashMap substitution)
    {
      if (substitution.isEmpty())
        return;

      int index = 0;
      int size = _raw.expressions.size();

      while (index < size) // skip to the first desugared pattern
        {
          Parameter parameter = (Parameter)_raw.patterns.get(index);
          if (parameter != null && parameter.isInternal())
            break;
          index++;
        }

      HashMap partialSubstitution = new HashMap(substitution.size());
      for (int i=index; i < size; i++)
```

```
      {
        _updateSubstitution(i,partialSubstitution,substitution);
        _raw.expressions.set(i,((Expression)_raw.expressions.get(i))
                             .substitute(partialSubstitution));
      }

    _raw.expression = _raw.expression.substitute(partialSubstitution);
  }
```

Adds to the specified partial substitution any expression from `reference` an indexed-expression substitution) with an index less than, or equal to, the specified index, removing such indexed-expressions from `reference`.

```
private final static void _updateSubstitution (int index, HashMap partial, HashMap refere
  {
    ArrayList keys = new ArrayList();

    for (Iterator i=reference.entrySet().iterator(); i.hasNext();)
      {
        Map.Entry entry = (Map.Entry)i.next();
        String key = (String)entry.getKey();
        IndexedExpression value = (IndexedExpression)entry.getValue();
        if (index <= value.index)
          {
            partial.put(key,value.expression);
            keys.add(key);
          }
      }

    int size = keys.size();
    for (int i=size; i-->0;)
      reference.remove(keys.get(i));
  }
```

First unnests the filters of all nested comprehensions if any, then unnests the filters of this comprehension. It is important to proceed bottom up because filters may migrate up from inner comprehensions, and therefore the filters of a comprehension must be unnested only after those of its nested comprehensions have been unnested.

```
final void unnestInnerFilters ()
  {
    if (_nestedComprehensionCount > 0)
      {
        _raw.expression.unnestInnerFilters();
        for (int i=_raw.expressions.size(); i-->0;)
```

```
          ((Expression)_raw.expressions.get(i)).unnestInnerFilters();
        }

    _unnestFilters();
  }
```

Normalizes the qualifiers of this comprehension by unnesting the filters to the left as far as they may go, recognizing selectors and slicing filters, and sets the construct to the translation of the comprehension

```
private final void _unnestFilters ()
  {
    Qualifier[] qualifiers = new Qualifier[_raw.patterns == null ? 0 : _raw.patterns.size
    for (int i=qualifiers.length; i-->0;)
      qualifiers[i] = new Qualifier((Parameter)_raw.patterns.get(i),
                                    (Expression)_raw.expressions.get(i));

    if (qualifiers.length > 0) _normalize(qualifiers);

    _construct = _translate(qualifiers,0);
    if (_doLetWrapping && !_isLetWrapped())
      {
        Parameter[] monoidParameters = { new Parameter("$OP$"), new Parameter("$ID$") };
        Expression[] monoidComponents = { _operation, _identity };

        _construct = new Let(monoidParameters,monoidComponents,_construct);
      }

    _raw = null;
    //Debug.step(this);
  }
```

Returns `true` iff the first comprehension in which this is nested (if any) is one involving the same monoid—then, as it is already wrapped inside that comprehension `Let` over the same operation and identity, there is no needed to wrap it again.

```
private final boolean _isLetWrapped ()
  {
    for (Expression e = _enclosingScope; e != null; e = e.enclosingScope())
      if (e instanceof Comprehension)
        {
          Comprehension c = (Comprehension)e;
          if (operation().equals(c.operation()) && identity().equals(c.identity()))
            return true;
```

```
          return false;
        }

    return false;
  }
```

Reshapes the specified array of qualifiers unnesting all boolean filters by moving them to the left as far as they may go (*i.e.*, no further than a generator whose parameter occurs free in the filter), and merging all filters related to the same generator into an common `and`. A selector or slicing condition is recognized and treated specially: it is passed to its generator qualifier where it is then processed appropriately.

```
private final void _normalize (Qualifier[] qualifiers)
  {
    //System.out.print("Before normalization..."); Debug.step(qualifiers);
    _unnestFilters(qualifiers.length-1,qualifiers);
    //System.out.print("After normalization...");  Debug.step(qualifiers);
  }
```

Normalizes the specified array of qualifiers up to the specified index minus one, then proceeds to unnest leftward the qualifier at the specified index.

```
private final void _unnestFilters (int index, Qualifier[] qualifiers)
  {
    if (index == -1) return;

    int upperLimit = index;
    _unnestFilters(upperLimit-1,qualifiers);
    Qualifier qualifier = qualifiers[index];

    // push this qualifier to the left over null qualifiers if any
    int i = index-1;
    while (i >= 0 && qualifiers[i] == null) i--;
    if (i < index-1)
      {
        qualifiers[index] = null;
        qualifiers[index = i+1] = qualifier;
      }

    if (qualifier.isGenerator()) return;

    // this qualifier is then a filter - unnest it as far as it can go
    while (index > 0)
      if (qualifiers[index-1].isGenerator())
        if (qualifier.expression.containsFreeName(qualifiers[index-1].parameter.name()))
```

```
        { // collect if selector, or slicing with no selectors; else, leave the filter
          if (qualifier.isSelector(qualifiers[index-1].parameter))
            {
              qualifiers[index-1].addSelector(qualifier.expression);
              _eraseQualifier(index,upperLimit,qualifiers);
            }
          else
            if (qualifiers[index-1].selectors == null
                && qualifier.isSlicing(qualifiers[index-1].parameter))
              {
                qualifiers[index-1].addSlicing(qualifier.expression);
                _eraseQualifier(index,upperLimit,qualifiers);
              }
          return; // this is as far as it can go
        }
      else // move this filter over one step to the left
        {
          qualifiers[index] = qualifiers[index-1];
          qualifiers[index = index-1] = qualifier;
        }
    else // qualifiers[index-1] is a filter
      if (index > 1) // if qualifiers[index-2] exists, it must contain a generator
        if (!qualifier.expression.containsFreeName(qualifiers[index-2].parameter.name(
          { // move this filter over two steps to the left
            qualifiers[index] = qualifiers[index-1];
            qualifiers[index-1] = qualifiers[index-2];
            qualifiers[index = index-2] = qualifier;
          }
        else // collect if selector, or slicing with no selectors; else, merge into the
          {
            if (qualifier.isSelector(qualifiers[index-2].parameter))
              qualifiers[index-2].addSelector(qualifier.expression);
            else
              if (qualifiers[index-2].selectors == null
                  && qualifier.isSlicing(qualifiers[index-2].parameter))
                qualifiers[index-2].addSlicing(qualifier.expression);
              else // merge this filter with the previous one using an 'and'
                qualifiers[index-1].expression = new And(qualifiers[index-1].expressio
                                                         qualifier.expression);
            _eraseQualifier(index,upperLimit,qualifiers);
            return; // this is as far as it can go
          }
      else // unnest further up, or merge this filter into the previous one using an 'a
        {
          if (!_isFurtherUnnestable(qualifier.expression))
            qualifiers[index-1].expression = new And(qualifiers[index-1].expression,
                                                     qualifier.expression);
          _eraseQualifier(index,upperLimit,qualifiers);
```

```
            return; // this is as far as it can go
          }

      // index == 0
      if (_isFurtherUnnestable(qualifier.expression))
        _eraseQualifier(index,upperLimit,qualifiers);
    }
```

Goes up the scope tree as far as it can without crossing a scope that either contains more than one nested comprehension or captures a free variable in the specified filter, until it reaches a comprehension. If it can do so and the found comprehension is of same nature as this one, adds the filter to that comprehension, and returns `true`; otherwise, returns `false`.

```
  private final boolean _isFurtherUnnestable (Expression filter)
    {
      Expression enclosingScope = _enclosingScope;

      while (enclosingScope != null && enclosingScope.nestedComprehensionCount() == 1)
        {
          if (enclosingScope instanceof Comprehension)
            {
              Comprehension comp = (Comprehension)enclosingScope;
              if (operation().equals(comp.operation()) && identity().equals(comp.identity(
                {
                  comp.addFilter(filter);
                  return true;
                }

              return false;
            }

          Scope scope = (Scope)enclosingScope;
          for (int i=scope.arity(); i-->0;)
            if (filter.containsFreeName(scope.parameter(i).name()))
              return false;

          enclosingScope = scope.enclosingScope();
        }

      return false;
    }
```

Adds the specified filter to the qualifiers of this comprehension.

```
final void addFilter (Expression filter)
    {
```

```
      _raw.patterns.add(null);
      _raw.expressions.add(filter);
    }
```

Sets the qualifier at the specified `index` to `null` and percolates this `null` as far to the right as it may go.

```
  private final static void _eraseQualifier (int index, int upperLimit, Qualifier[] qualifi
    {
      qualifiers[index] = null;
      for (int i=index; i < upperLimit && qualifiers[i+1] != null; i++)
        {
          qualifiers[i] = qualifiers[i+1];
          qualifiers[i+1] = null;
        }
    }
```

This translates monoid comprehension syntax using (possibly filtered) homomorphisms. It assumes that the array of qualifiers has been normalized. The translation scheme is as follows:

```
[op,id]{e | } = op(e,id);
[op,id]{e | c} = if c then op(e,id) else id;
[op,id]{e | x <- e', c, Q} = f_hom(e', lambda x.[op,id]{e | Q}, op, id, lambda x.c);
[op,id]{e | x <- e', y <- e'', Q} = hom(e', lambda x.[op,id] { e| y <- e'', Q}, op, id);
```

```
  private final Expression _translate (Qualifier[] qualifiers, int index)
    {
      if (index == qualifiers.length || qualifiers[index] == null)
        return new Application(_raw.op(),_raw.expression,_raw.id());

      Expression body = null;
      Homomorphism hom = null;

      if (index < qualifiers.length-1 && qualifiers[index].parameter != null
          && qualifiers[index+1] != null && qualifiers[index+1].parameter == null)
        {
          body = _translate(qualifiers,index+2);

          if (qualifiers[index].selectors != null)
            return _selectorExpression(qualifiers[index],qualifiers[index+1].expression,bo

          hom = new FilterHomomorphism(_tables,
                                       qualifiers[index].expression,
                                       new Scope(qualifiers[index].parameter,body),
                                       _raw.op(),_raw.id(),
                                       new Scope((Parameter)qualifiers[index].parameter.ty
```

```
                                                    qualifiers[index+1].expression));
        }
      else
        {
          body = _translate(qualifiers,index+1);

          if (qualifiers[index].parameter == null)
            return new IfThenElse(qualifiers[index].expression,body,_raw.id());

          if (qualifiers[index].selectors != null)
            return _selectorExpression(qualifiers[index],null,body);

          hom = new Homomorphism(qualifiers[index].expression,
                                 new Scope(qualifiers[index].parameter,body),
                                 _raw.op(),_raw.id());
        }

      if (qualifiers[index].slicings != null)
        hom.setSlicings(qualifiers[index].slicings);

      if (_raw.inPlace == Homomorphism.ENABLED_IN_PLACE)
        return hom.enableInPlace();
      if (_raw.inPlace == Homomorphism.DISABLED_IN_PLACE)
        return hom.disableInPlace();

      return hom;
    }
```

This returns a `Let` wrapping an `IfThenElse` as the transformed expression resulting from a (possibly filtered) generator that contains at least one selector expression. More precisely, if the generator is of the form:

```
x <- e such that f
       sliced by s1, ..., sm
       selected by v1, ..., vn
```

and the body of translating the remaining qualifiers is `body`, then the resulting selector expression is:

```
let x = v1
    in if x is_in e
          and x == v2 and ... and x == vn
          and s1 and ... and sm
          and f
       then body
       else id
```

where each slicing has its slicing variable unsanitized from a dummy local back to a dummy.

```
  private final Expression _selectorExpression (Qualifier generator, Expression filter,
                                                Expression body)
    {
      Expression condition = new Application(_tables.in(),
                                             new Dummy(generator.parameter),
                                             generator.expression);

      for (int i=1; i<generator.selectors.size(); i++)
        condition = new And(condition,
                            (Expression)generator.selectors.get(i));

      if (generator.slicings != null)
        for (int i=0; i<generator.slicings.size(); i++)
          condition = new And(condition,
                              ((Application)generator.slicings.get(i)).undoDummyLocal());

      if (filter != null)
        condition = new And(condition,filter);

      return new Let(generator.parameter,
                     ((Application)generator.selectors.get(0)).argument(1),
                     new IfThenElse(condition,body,_raw.id()));
    }


  public final void setCheckedType ()
    {
      if (setCheckedTypeLocked()) return;
      _construct.setCheckedType();
      setCheckedType(_construct.checkedType());
    }


  public final void typeCheck (TypeChecker typeChecker) throws TypingErrorException
    {
      if (typeCheckLocked()) return;

      if (!(_construct instanceof Let))
        {
          _construct.typeCheck(_type,typeChecker);
          return;
        }

      Let let = (Let)_construct;
      let.setType(_type);

      Scope scope = (Scope)let.function();
      typeChecker.unify(scope.parameter(0).typeRef(),operation().typeRef(),this);
      typeChecker.unify(scope.parameter(1).typeRef(),identity().typeRef(),this);
```

```
    Expression operation = let.argument(0);
    Expression identity = let.argument(1);

    Type[] argumentTypes = { operation.typeRef(), identity.typeRef() };
    FunctionType functionType = new FunctionType(argumentTypes,let.typeRef()).setNoCurry

    identity.typeCheck(typeChecker);
    let.function().typeCheck(functionType,typeChecker);
    operation.typeCheck(functionType.domains()[0],typeChecker);
  }

public final void compile (Compiler compiler)
  {
    if (_construct instanceof Let) _fixTypeBoxing();
    _construct.compile(compiler);
  }
```

This fixes the boxing of the monoid operator and identity by systematically unboxing all occurrences of the collection element type. This is necessary because collection-building built-in dummy instructions like SET_ADD have a needlessly polymorphic type that becomes instantiated only when it is applied. However, a monoid comprehension construct is a Let 3.3.12 that abstracts the monoid operation and identity. Now, when the operation is SET_ADD, for example, as the compiler compiles the application corresponding to the "Let," it sees it as a non-applied function argument with a polymorphic type and will proceed to "pad" it (see Expression 3.1. This "padding" must be avoided, as well as all boxing of the types corresponding to the elements of the collection.

```
private final void _fixTypeBoxing ()
  {
    Let let = (Let)_construct;

    FunctionType potype = (FunctionType)((FunctionType)let.function().checkedType()).doma
    FunctionType otype = (FunctionType)let.argument(0).checkedType();
    Type itype = let.argument(1).checkedType();

    if (itype.kind() == Type.BOXABLE)
      ((BoxableTypeConstant)itype).setBoxed(false);

    if (otype.domain(0).kind() == Type.BOXABLE)
      {
        ((BoxableTypeConstant)otype.domain(0)).setBoxed(false);
        otype.unsetDomainBox(0);

        ((BoxableTypeConstant)potype.domain(0)).setBoxed(false);
        potype.unsetDomainBox(0);
```

```
        if (otype.domain(0).isEqualTo(otype.domain(1)))   // primitive comprehension
          {
            ((BoxableTypeConstant)otype.domain(1)).setBoxed(false);
            otype.unsetDomainBox(1);

            ((BoxableTypeConstant)potype.domain(1)).setBoxed(false);
            potype.unsetDomainBox(1);

            ((BoxableTypeConstant)otype.range()).setBoxed(false);
            otype.unsetRangeBox();

            ((BoxableTypeConstant)potype.range()).setBoxed(false);
            potype.unsetRangeBox();
          }
      }
  }

public final String toString ()
  {
    return _raw == null ? _construct.toString() : _raw.toString();
  }

public final String toTypedString ()
  {
    return _raw == null ? _construct.toString() : _raw.toString() + " : " +
           checkedType() == null ? type().toString() : checkedType().toString();
  }

private class RawInfo
  {
    Expression operation;
    Expression identity;
    Expression expression;
    AbstractList patterns;
    AbstractList expressions;
    byte inPlace;

    boolean isDesugared;

    RawInfo (Expression operation, Expression identity, Expression expression,
             AbstractList patterns, AbstractList expressions, byte inPlace)
      {
        this.expression = expression;
        this.operation = operation;
        this.identity = identity;
        this.patterns = patterns;
```

```
        this.expressions = expressions;
        this.inPlace = inPlace;
      }

    final Expression op ()
      {
        return operation.typedCopy();
      }

    final Expression id ()
      {
        return identity.typedCopy();
      }

    public final String toString ()
      {
        StringBuffer buf = new StringBuffer("[")
                                .append(operation())
                                .append(",")
                                .append(identity())
                                .append("] { ")
                                .append(expression)
                                .append(" | ");

        for (int i=0; i<patterns.size(); i++)
          {
            Object pattern = patterns.get(i);
            if (pattern != null)
              buf.append(pattern).append(" <- ");
            buf.append(expressions.get(i));
            if (i < patterns.size() - 1)
              buf.append(", ");
          }

        return buf.append(" }").toString();
      }
  }

private static class IndexedExpression
  {
    int index = -1;
    Expression expression;

    IndexedExpression (Expression expression)
      {
        this.expression = expression;
      }
```

```
    IndexedExpression (int index, Expression expression)
      {
        this.index = index;
        this.expression = expression;
      }

    public final String toString ()
      {
        return expression + "/" + index;
      }
  }

private class Qualifier
  {
    Parameter parameter;
    Expression expression;
    ArrayList slicings;
    ArrayList selectors;

    Qualifier (Parameter parameter, Expression expression)
      {
        this.parameter = parameter;
        this.expression = expression;
      }

    final boolean isGenerator ()
      {
        return parameter != null;
      }

    final boolean isSlicing (Parameter parameter)
      {
        return expression.isSlicing(tables(),parameter);
      }

    final void addSlicing (Expression slicing)
      {
        if (slicings == null)
          slicings = new ArrayList();
        slicings.add(slicing);
      }

    final boolean isSelector (Parameter parameter)
      {
        return expression.isSelector(tables(),parameter);
      }

    final void addSelector (Expression selector)
```

```
        {
          if (selectors == null)
            selectors = new ArrayList();
          selectors.add(selector);
        }

      public final String toString ()
        {
          if (parameter == null)
            return expression.toString();

          return parameter + " <- " + expression +
                 (selectors == null ? "" : " selected by " + selectors) +
                 (slicings == null ? "" : " sliced by " + slicings);
        }
    }
}
```

Chapter 4

The type language

4.1 Overview

We first define some basic terminology regarding the type system and operations on types.

4.1.1 Polymorphism

Here, by "*polymorphism*," we mean ML-polymorphism (*i.e.*, 2*nd*-order universal)—with a few differences that will be explained along the way—in other words, types presented with a grammar such as:

[1] *Type* ::= *SimpleType* | *TypeScheme*

[2] *SimpleType* ::= *BasicType* | *FunctionType* | *TypeParameter*

[3] *BasicType* ::= $\mathfrak{Int}$ | $\mathfrak{Real}$ | $\mathfrak{Boolean}$ | ...

[4] *FunctionType* ::= $SimpleType \rightarrow SimpleType$

[5] *TypeParameter* ::= α | α' | ... | β | β' | ...

[6] *TypeScheme* ::= $\forall$ *TypeParameter* . *Type*

that ensures that universal type quantifiers occur only at the outset of a polymorphic type.[1]

[1] Or more precisely that $\forall$ never occurs nested inside a function type arrow $\rightarrow$. This apparently innocuous detail ensures decidability of type inference. BTW, the 2*nd* order comes from the fact that the quantifier applies to *type* parameters (as opposed to 1*st* order, if it had applied to *value* parameters). The *universal* comes from $\forall$, of course.

4.1.2 Multiple Type Overloading

This is also often called *ad hoc* polymorphism. When enabled (the default), this allows a same identifier to have several unrelated types. Generally, it is restricted to names with functional types. However, since functions are first-class citizens, this restriction makes no sense, and therefore the default is to enable multiple type overloading for all types.

Note that there is no established technology that prevails for supporting *both* ML-polymorphic type inference and multiple type overloading. Here (and in several other parts of this overall design) I have had to innovate and put to use techniques from (Constraint) Logic Programming to be able to prove the combination of types supportable by this architecture.

4.1.3 Currying

Currying is an operation that exploits the following mathematical isomorphism of types:[2]

$$T, T' \rightarrow T'' \simeq T \rightarrow (T' \rightarrow T'') \tag{4.1}$$

which can be generalized to its multiple form:

$$T_1, \ldots, T_n \rightarrow T \simeq T_1, \ldots, T_k \rightarrow (T_{k+1}, \ldots, T_n \rightarrow T) \quad k = 1, \ldots, n-1 \tag{4.2}$$

When function currying is enabled, this means that type-checking/inference must build this equational theory into the type unification rules in order to consider types equal modulo this isomorphism.

4.1.4 Standardizing

As a result of, *e.g.*, currying, the shape of a function type may change in the course of a type-checking/inference process. Type comparison may thus be tested on various structurally different, although syntactically congruent, forms of a same type. A type must therefore assume a canonical form in order to be compared. This is what *standardizing* a type does.

Standardizing is a two-phase operation that first *flattens* the domains of function types, then *renames* the type parameters. The flattening phase simply amounts to uncurrying as much as possible by applying Equation (4.1) as a rewrite rule, although *backwards* (*i.e.*, from right to left) as

[2] For the intrigued reader curious to know what deep connection there might be between functional types and Indian cooking, the answer is, *"None whatsoever!"* The word was coined after Prof. Haskell B. Curry's last name. Curry was one of the two mathematicians/logicians (along with Robert Feys) who conceived *Combinator Logic* and *Combinator Calculus*, and made extensive use of the isomorphism of Equation (4.1)—hence the folklore's use of the verb *to curry—(currying, curryed),*— in French: *curryfier—(curryfication, curryfié)*, to mean transforming a function type of several arguments into that of a function of one argument. The homonymy is often amusingly mistaken for an exotic way of [un]spicing functions.

long as it applies. The second phase (renaming) consists in making a consistent copy of all types reachable from a type's root.

4.1.5 Copying

Copying a type is simply taking a duplicate twin of the graph reachable from the type's root. Sharing of pointers coming from the fact that type parameters co-occur are recorded in a parameter substitution table (in our implementation, simply a `java.util.HashMap`) along the way, and thus consistent pointer sharing can be easily made effective.

4.1.6 Equality

Testing for equality must be done modulo a parameter substitution table (in our implementation, simply a `java.util.HashMap`) that records pointer equalities along the way, and thus equality up to parameter renaming can be easily made effective.

A tableless version of equality also exists for which each type parameter is considered equal only to itself.

4.1.7 Unifying

Unifying two types is the operation of filling in missing information (*i.e.*, type parameters) in each with existing information from the other by side-effecting (*i.e.*, binding) the missing information (*i.e.*, the type parameters) to point to the part of the existing information from the other type they should be equal to (*i.e.*, their values). Note that, like logical variables in Logic Programming, type parameters can be bound to one another and thus must be dereferenced to their values.

4.1.8 Boxing/Unboxing

The kernel language is polymorphically typed. Therefore, a function expression that has a polymorphic type must work for all instantiations of this type's type parameters into either primitive unboxed types (*e.g.*, $\mathfrak{Int}$, $\mathfrak{Real}$, *etc.*) or boxed types. The problem this poses is: how can we compile a polymorphic function into code that would correctly know what the actual runtime sorts of the function's runtime arguments and returned value are, *before the function type is actually instantiated into a (possibly monomorphic) type*?[3] The problem was addressed by Xavier Leroy

[3]Besides compiling distinct copies for all possible runtime sort instantiations (like, *e.g.*, C++ template functions), nor recompiling each time a specific instantiation is needed. The former is not acceptable because its tends to inflate the code space explosively. The latter can neither be envisaged because it goes against a few (rightfully) sacrosanct

10 years ago [5] and he proposed a solution.[4] Leroy's method is based on the use of type annotation that enables a source-to-source transformation. This source transformation is the automatic generation of *wrappers* and *unwrappers* for boxing and unboxing expressions whenever necessary. After that, compiling the transformed source as usual will be guaranteed to be correct on all types.

I adapted and improved the main idea from Leroy's solution so that:

- the type annotation and rules are greatly simplified;
- no source-to-source transformation is needed;
- un/wrappers generation is done at code-generation time.

This saves a great amount of space and time.

4.2 The type system

The type system consists of two complementary parts: a *static* and a *dynamic* part.[5] The former takes care of verifying all type constraints that are statically decidable (*i.e.*, before actually running the program). The latter pertains to type constraints that must wait until execution time to decide whether those (involving runtime values) may be decided. This is called dynamic type-checking and is best seen (and conceived) as an *incremental* extension of the static part.

A type is either a static type, or a dynamic type. A static type is a type that is checked before runtime by the type-checker. A dynamic type is a wrapper around a type that may need additional runtime information in order to be fully verified. Its static part must be (and is!) checked statically by the static type checker, but the compiler may complete this by issuing runtime tests at adequate places in the code it generates; namely, when:

- binding abstraction parameters of this type in an application, or
- assigning to local and global variable of this type, or
- updating an array slot, a tuple component, or an object's field, of this type.

There are two kinds of dynamic types:

- Extensional types—defined with explicit extensions (either statically provided or dynamically computed runtime values):

principles like separate compilation and abstract library interfacing—imagine having to recompile code from a library everytime you want to use it!

[4] This solution is the one implemented in the CAML compiler [6].

[5] See Appendix Section B.2 on Page 80 for the complete class hierarchy of types in the package `ilog.language.design.types`.

  - Set extension type;
  - Int range extension type (close interval of ints);
  - Real range extension type (close interval of reals).

  A special kind of set of int type is used to define enumeration types (from actual symbol sets) through opaque type definitions.

- Intensional types—defined using any runtime boolean condition to be checked at runtime, calls to which are tests generated statically; *e.g.*non-negative numbers (*i.e.*, `int+`, `float+`).

4.3 Static types

The static type system...

4.3.1 Primitive types

Boxable types

- Void
- Int
- Real
- Char
- Boolean

Boxed types

Built-in type constants (*e.g.*, String).

4.3.2 Type constructors

Function types

Tuple types

Position tuple types

Named tuple types

Array types

0-based int-indexed arrays

Int range-indexed arrays

Set-indexed arrays

Multidimensional arrays

Collection types

Set, bag, and list types

Class types

This is the type of object structures. It declares an *interface* (or member type signature) for a class of objects and the members comprising its structure. It holds information for compiling field access and update, and enables specifying an *implementation* for methods manipulating objects of this type.

A class implementation uses the information declared in its interface. It is interpreted as follows: only non-method members—hereafter called *fields*—correspond to actual slots in an object structure that is an instance of the class and thus may be updated. On the other hand, all members (*i.e.*, both fields and method members) are defined as global *functions* whose first argument stands for the object itself (that may be referred to as '$\mathfrak{this}$').

The syntax we shall use for a class definition is of the form:

$$\mathfrak{class}\ \texttt{classname}\ \{\ \textit{interface}\ \}\ [\ \{\ \textit{implementation}\}\ ] \tag{4.3}$$

The *interface* block specifies the type signatures of the *members* (*fields* and *methods*) of the class and possibly initial values for fields. The *implementation* block is optional and gives the definition of (some or all of) the methods.

For example, one can declare a class to represent a simple counter as follows:

$$\begin{array}{l}\mathfrak{class}\ \texttt{Counter}\ \{\ \texttt{value} : \mathfrak{Int} = 1;\\ \qquad\qquad\quad\ \ \mathfrak{method}\ \texttt{set} : \mathfrak{Int} \rightarrow \texttt{Counter};\\ \qquad\qquad\quad\ \}\\ \qquad\qquad\quad\ \{\ \texttt{set}(\texttt{value} : \mathfrak{Int}) : \texttt{Counter} = (\mathfrak{this}.\texttt{value} = \texttt{value});\\ \qquad\qquad\quad\ \}\end{array} \tag{4.4}$$

The first block specifies the interface for the class type `Counter` defining two members: a field `value` of type $\mathfrak{Int}$ and a method `set` taking an argument of type $\mathfrak{Int}$ and returning a `Counter`

object. It also specifies an initialization expression (1) for the `value` field. Specifying a field's initialization is optional—when missing, the field will be initialized to a null value of appropriate type: 0 for an $\mathfrak{Int}$, 0.0 for a $\mathfrak{Real}$, $\mathfrak{false}$ for a $\mathfrak{Boolean}$, $'\backslash 000'$ for a $\mathfrak{Char}$, `""` for a $\mathfrak{String}$, $\mathfrak{void}$ for $\mathfrak{Void}$,[6] and $\mathfrak{null}_T$ for any other type T. The implementation block for the `Counter` class defines the body of the `set` method. Note that a method's implementation can also be given outside the class declaration as a function whose first argument's type is the class. For example, we could have defined the `set` method of the class `Counter` as:

$$\mathfrak{def}\ \texttt{Counter::set(x : Counter, n : Int) : Counter} = (\texttt{x.value} = \texttt{n}); \tag{4.5}$$

On the other hand, although a field is also semantically a function whose first argument's type is a class, it may *not* be defined outside its class. Defining a declared field outside a class declaration causes an error. This is because the code of a field is always fixed and defined to return the value of an object's slot corresponding to the field. Note however that one may define a unary function whose argument is a class type outside this class when it is not a declared field for this class. It will be understood as a *method* for the class (even though it takes no extra argument and may be invoked in "dot notation" without parentheses as a field is) and thus act as a "static field" for the class. Of course field updates using dot notation will not be allowed on these pseudo fields. However, they (like any global variable) may be (re)set using a global (re)definition at the top level, or a nested global assignment.

Note also that a field may be functional without being a method—the essential difference being that a field is part of the structure of every object instance of a class and thus may be updated within an object instance, while a method is common to all instances of a class and may not be updated within a particular instance, but only globally for all the class' instances.

Thus, everytime a `Counter` object is created with $\mathfrak{new}$, as in, for example:

$$\texttt{c} = \mathfrak{new}\ \texttt{Counter}; \tag{4.6}$$

the value 1 will be used to initialize the slot that corresponds to the location of the `value` field. Then, field and method invocation can be done using the familiar "dot notation"; *e.g.*:

```
c.set(c.value + 2);
write(c.value);
```
(4.7)

This will set `c`'s `value` field to 3 and print out this value. This code is exactly equivalent to:

```
Counter::set(c, Counter::value(Counter::c) + 2);
write(Counter::value(Counter::c));
```
(4.8)

[6] Strictly speaking, a field of type $\mathfrak{Void}$ is useless since it can only have the unique value of this type (*i.e.*, $\mathfrak{void}$). Thus, a $\mathfrak{void}$ field should arguably be disallowed. On the other hand, allowing it is not semantically unsound and may be tolerated for the sake of uniformity.

Indeed, field and method invocation simply amounts to functional application. This scheme offers the advantage that an object's fields and methods may be manipulated as functions (*i.e.*, as first-class citizens) and no additional setup is needed for type-checking and/or type inference when it comes to objects.

Incidentally, some or all type information may be omitted while specifying a class's *implementation* (though not its *interface*) as long as non-ambiguous types may be inferred. Thus, the implementation block for class `Counter` in class definition (4.4) could be written more simply as:

```
{ set(n) = (value = n); }
```
(4.9)

Declaring a class type and defining its implementation causes the following:

- the name of the class is entered with a new type for it in the type table (an object comprising symbol tables, of type `ilog.language.design.types.Tables`, where its type definition associates it with a `ClassType` whose class structure is encapsulated by an object of type `ilog.language.design.types.ClassInfo` where code entries for all its members' types are recorded;
- each field of a distinct type is assigned an offset in an array of slots (per sort);
- each method and field expression is name-sanitized, type-checked, and sort-sanitized after closing it into an abstraction taking this as first argument;
- each method definition is then compiled into a global definition, and each field is compiled into a global function corresponding to accessing its value from the appropriate offset;
- finally, each field's initialization expression is compiled and recorded in the `ClassType` to be used at object creation time. An object may be created at run-time (using the new operator followed by a class name).

4.3.3 Polymorphic types

4.4 Type definitions

Before we review dynamic types, we shall describe how one can define new types using existing types. Type definitions are provided both for convenience of making programs more legible by giving "logical" names (or terms) to otherwise verbose types, and that of hiding information details of a type making it act as a new type altogether. The former facility is that of providing *aliases* to types (exactly like a preprocessor's macros get expanded right away into their textual equivalents), while the latter offers the convenience of defining *new* types in terms of existing ones, but hiding this information. It follows from this distinction that a type alias is *always* structurally equivalent

to its value (in fact an alias disappears as soon as it is read in, being parsed away into the structure defining it). By contrast, a defined type is *never* structurally equivalent to its value nor any other type—it is only equivalent to itself. To enable meaningful computation with a defined type, two meta-(de/con)structors are thus provided: one for explicitly *casting* a defined type into the type that defines it, and one explicitly seeing a type as a specified defined type (if such a defined type does exist and with this type as definition).

The class `ilog.language.design.types.Tables` contains the symbol tables for global names and types. The name spaces of the identifiers denoting type and non-type (global or local) names (which are kept in the global symbol table) are disjoint—so there are no name conflicts between types and non-type identifiers.

The `typeTable` variable contains the naming table for types and the `symbolTable` variable contains the naming table for other (non-type) global names.

This section will unfold all the type-related data-structures starting from the class that manages symbols: `ilog.language.design.types.Tables`. The names can be those of types and values. They are *global* names.[7] The type namespace is independent of the value namespace—*i.e.*, the same name can denote a value and a type.

4.4.1 Type aliasing

4.4.2 Type hiding

4.5 Dynamic types

Dynamic types are to be checked, if possible statically (at least their static part is), at least in two particular places of an expression. Namely,

- at assignment/update time; and,
- at (function) parameter-binding time.

This will ensure that the actual value placed in the slot expecting a certain type does respect additionnal constraints that may only be verified with some runtime values. Generally, dynamic types are so-called *dependent* types (such as, *e.g.*, `array_of_size`(n), a "safe" array type depending on the array size that may be only computed at runtime—*i.e.*, *à la* Java arrays.).

From this, we require that a class implementing the `DynamicType` interface provides a method `public boolean verifyCondition()` that is invoked systematically by code generated

[7] At the moment, there is no name qualification or namespace management. When this service is provided, it will also be through the `ilog.language.design.types.Tables` class.

for dynamically typed function parameters and for locations that are the target of updates (*i.e*, array slot update, object field update, tuple field update) at compilation of abstractions and various assignment constructs. Of this class, three subclasses derive their properties:

- extensional types;
- Boolean-assertion types;
- non-negative number types.

We shall consider here a few such dynamic types (motivated esssentially by the need expressed for OPL, and hence NGO, types). Namely,

- extensional types;
- intensional types (*e.g.*, non-negative numbers)

An *extensional* type is a type whose elements are determined to be members of a predetermined and fixed extension (*i.e.*, any runtime value that denotes a collection—such as a set, an int range, a float range, or an enumeration). Such types pose the additional problem of being usable at compile-time to restrict the domains of other variables. However, some of those variables' values may only fully be determined at runtime. These particular dynamic types have therefore a simple `verifyCondition()` method that is automatically run as soon as the extension is known. It just verifies that the element is a *bona fide* member of the extension), otherwise it relies on a more complicated scheme based on the notion of *contract*. Basically, a contract-based type is an extensional type that does not have an extension (as yet) but already carries the obligation that some particular individual constants be part of their extensions. Those elements consitute "contracts" that must be honored as soon as the type's extension becomes known (either positively—eliminating the contract, or negatively—causing a type error).

4.5.1 Extensional types

Set types

Int range types

Float range types

Enum types

4.5.2 Intensional types

Example: non-negative numbers

Define new (opaque) types `Nat` as a dynamically constrained $\mathfrak{Int}$ type...

Chapter 5

The intermediate language

The complete list of instructions that are currently defined is as follows.

5.1 Do-nothing instruction

1. NO_OP

5.2 Push instructions

1. PUSH_I
2. PUSH_O
3. PUSH_R
4. PUSH_OFFSET_I
5. PUSH_OFFSET_O
6. PUSH_OFFSET_R
7. PUSH_TUPLE
8. PUSH_SET_I
9. PUSH_SET_R
10. PUSH_SET_O
11. PUSH_INT_RNG
12. PUSH_REAL_RNG

13. PUSH_CLOSURE
14. PUSH_NEW_OBJECT

5.3 Subroutine instructions

1. APPLY
2. APPLY_HOM_I
3. APPLY_HOM_R
4. APPLY_HOM_O
5. APPLY_IP_HOM_I
6. APPLY_IP_HOM_R
7. APPLY_IP_HOM_O
8. APPLY_COLL_I
9. APPLY_COLL_R
10. APPLY_COLL_O
11. APPLY_COLL_HOM_I
12. APPLY_COLL_HOM_R
13. APPLY_COLL_HOM_O
14. APPLY_IP_COLL_HOM_I
15. APPLY_IP_COLL_HOM_R
16. APPLY_IP_COLL_HOM_O
17. CALL
18. END
19. RETURN_I
20. RETURN_R
21. RETURN_O
22. NL_RETURN_I
23. NL_RETURN_R
24. NL_RETURN_O

5.4 Pop instructions

1. POP_I
2. POP_O
3. POP_R

5.5 Relocatable instructions

1. JUMP
2. JUMP_ON_FALSE
3. JUMP_ON_TRUE

5.6 Conversion instructions

1. I_TO_O
2. I_TO_R
3. O_TO_I
4. O_TO_R
5. R_TO_I
6. R_TO_O
7. ARRAY_TO_MAP_I
8. ARRAY_TO_MAP_R
9. ARRAY_TO_MAP_O
10. MAP_TO_ARRAY_O
11. CHECK_ARRAY_SIZE
12. RECONCILE_INDEXABLES
13. ARRAY_INITIALIZE
14. SHUFFLE_MAP_I
15. SHUFFLE_MAP_R
16. SHUFFLE_MAP_O

5.7 Assignment instructions

1. SET_GLOBAL
2. SET_OFFSET_I
3. SET_OFFSET_O
4. SET_OFFSET_R

5.8 Tuple component instructions

1. GET_TUPLE_I
2. GET_TUPLE_R
3. GET_TUPLE_O
4. SET_TUPLE_I
5. SET_TUPLE_R
6. SET_TUPLE_O

5.9 Array/Map allocation instructions

1. PUSH_ARRAY_I
2. PUSH_ARRAY_R
3. PUSH_ARRAY_O
4. PUSH_MAP_I
5. PUSH_MAP_R
6. PUSH_MAP_O
7. MAKE_ARRAY_I
8. MAKE_ARRAY_R
9. MAKE_ARRAY_O
10. MAKE_MAP_I
11. MAKE_MAP_R
12. MAKE_MAP_O

13. FILL_ARRAY_IA
14. FILL_ARRAY_IM
15. FILL_ARRAY_OA
16. FILL_ARRAY_OM
17. FILL_ARRAY_RA
18. FILL_ARRAY_RM
19. FILL_MAP_IA
20. FILL_MAP_IM
21. FILL_MAP_OA
22. FILL_MAP_OM
23. FILL_MAP_RA
24. FILL_MAP_RM

5.10 Array/Map slot instructions

1. GET_ARRAY_I
2. GET_INT_INDEXED_MAP_I
3. GET_INT_INDEXED_MAP_O
4. GET_INT_INDEXED_MAP_R
5. GET_MAP_I
6. GET_ARRAY_O
7. GET_MAP_O
8. GET_ARRAY_R
9. GET_MAP_R
10. SET_ARRAY_I
11. SET_INT_INDEXED_MAP_I
12. SET_INT_INDEXED_MAP_O
13. SET_INT_INDEXED_MAP_R
14. SET_MAP_I
15. SET_ARRAY_O

16. SET_MAP_O
17. SET_ARRAY_R
18. SET_MAP_R

5.11 Field instructions

1. GET_FIELD_I
2. GET_FIELD_O
3. GET_FIELD_R
4. SET_FIELD_I
5. SET_FIELD_O
6. SET_FIELD_R

5.12 Built-in operations

5.12.1 Arithmetic operations

1. ADD_II
2. ADD_IR
3. ADD_RI
4. ADD_RR
5. SUB_II
6. SUB_IR
7. SUB_RI
8. SUB_RR
9. MINUS_I
10. MINUS_R
11. MUL_II
12. MUL_IR
13. MUL_RI

14. MUL_RR
15. DIV_II
16. DIV_IR
17. DIV_RI
18. DIV_RR
19. MODULUS
20. MIN_II
21. MIN_IR
22. MIN_RI
23. MIN_RR
24. MAX_II
25. MAX_IR
26. MAX_RI
27. MAX_RR
28. ABS_I_RI
29. ABS_R
30. SQRT
31. POWER

5.12.2 Arithmetic relations

1. EQU_II
2. EQU_OO
3. EQU_RR
4. NEQ_II
5. NEQ_OO
6. NEQ_RR
7. GTE_II
8. GTE_IR
9. GTE_RI

10. GTE_RR
11. GRT_II
12. GRT_IR
13. GRT_RI
14. GRT_RR
15. LTE_II
16. LTE_IR
17. LTE_RI
18. LTE_RR
19. LST_II
20. LST_IR
21. LST_RI
22. LST_RR

5.12.3 Boolean operations

1. NOT

5.12.4 Map and Size operations

1. MAP_SIZE
2. ARRAY_SIZE
3. INDEXABLE_SIZE
4. GET_INDEXABLE

5.12.5 Container operations

1. BELONGS_I
2. BELONGS_O
3. BELONGS_R

5.12.6 Set operations

1. SET_COPY
2. MAKE_SET_I
3. MAKE_SET_O
4. MAKE_SET_R
5. SET_DIFF
6. SET_SYM_DIFF
7. INTER
8. UNION
9. D_SET_DIFF
10. D_SET_SYM_DIFF
11. D_INTER
12. D_UNION

5.12.7 Set relations

1. SUBSET

5.12.8 Set element operations

1. SET_ADD_I
2. SET_ADD_R
3. SET_ADD_O
4. SET_RMV_I
5. SET_RMV_R
6. SET_RMV_O
7. FIRST_I
8. FIRST_O
9. FIRST_R
10. LAST_I
11. LAST_O

12. LAST_R
13. NEXT_I
14. NEXT_C_I
15. NEXT_O
16. NEXT_C_O
17. NEXT_R
18. NEXT_C_R
19. ORD_I
20. ORD_O
21. ORD_R
22. PREV_I
23. PREV_C_I
24. PREV_O
25. PREV_C_O
26. PREV_R
27. PREV_C_R

5.12.9 Range operations

1. INT_RNG_UB
2. INT_RNG_LB
3. REAL_RNG_UB
4. REAL_RNG_LB

5.12.10 String operations

1. STRCON

5.12.11 I/O operations

1. WRITE_I
2. WRITE_O
3. WRITE_R

5.13 Dummy instructions

1. DUMMY_EQU
2. DUMMY_NEQ
3. DUMMY_AND
4. DUMMY_OR
5. DUMMY_STRCON
6. DUMMY_WRITE
7. DUMMY_SIZE
8. DUMMY_SET_ADD
9. DUMMY_SET_RMV
10. DUMMY_BELONGS
11. DUMMY_ORD
12. DUMMY_FIRST
13. DUMMY_LAST
14. DUMMY_NEXT
15. DUMMY_NEXT_C
16. DUMMY_PREV
17. DUMMY_PREV_C

Chapter 6

The backend system

6.1 The runtime system

This is the class defining a runtime object. Such an object serves as the common execution environment context shared by `Instruction`s being executed. It encapsulates a state of comptutation that is effected by each instruction as it is executed in its context.

A `Runtime` object consists of attributes and structures that together define a state of computation, and methods that are used by instructions to effect this state as they are executed. Thus, each instruction class defines an `execute(Runtime)` method that specifies its operational semantics as a state transformation of its given runtime context.

Initiating execution of a `Runtime` object consists of setting its code array to a given instruction sequence, setting its instruction pointer `_ip` to its code's first instruction and repeatedly calling `execute(this)` on whatever instruction is currently at address `_ip` in the current code array. The final state is reached when a flag indicating that it is so is set to `true`. Each instruction is responsible for appropriately setting the next state according to its semantics, including saving and restoring states, and (re)setting the code array and the various runtime registers pointing into the state's structures.

Runtime states encapsulated by objects in this class are essentially those of a stack automaton, specifically conceived to support the computations of a higher-order functional language with lexical closures—*i.e.*, a λ-calculus machine—extended to support additional features—*e.g.*, assignment side-effects, objects, automatic currying... As such it may viewed as an optimized variant of Peter Landin's SECD machine [4]—in the same spirit as Luca Cardelli's Functional Abstract Machine (FAM) [1], although our design is quite different from Cardelli's in its structure and operations.

Because this is a Java implementation, in order to avoid the space and performance overhead of

being confined to boxed values for primitive type computations, three concurrent sets of structures are maintained: in addition to those needed for boxed (Java object) values, two extra ones are used to support unboxed integer and floating-point values, respectively. The runtime operations performed by instructions on a `Runtime` object are guaranteed to be type-safe in that each state is always such as it must be expected for the correct accessing and setting of values. Such a guarantee must be (and is!) provided by the `TypeChecker` and the `Sanitizer`, which ascertain all the conditions that must be met prior to having a `Compiler` proceed to generating instructions which will safely act on the appropriate stacks and environments of the correct sort (integer, floating-point, or object).

6.2 The runtime objects

6.3 The display manager

6.4 The error manager

Chapter 7

A full example—HAK_LL

This chapter details the design of a concrete language from scratch. We call this language HAK_LL—presumably to mean, somewhat presumptuously: Hassan Aït-Kaci's Little Language.[1]

HAK_LL is a fully-working prototype language whose essential goal is to illustrate and demonstrate our architecture: the expressive power of the kernel language and the workings of its type-checker, compiler, and runtime systems. It is an imperative functional language with objects, where functions are first-class citizens. HAK_LL has a surface syntax for an interactive language that can define top-level constructs and evaluate expressions. It supports 2nd-order (ML-like) type polymorphism, automatic currying, multiple type overloading, dynamic operator overloading, as well as flat classes and objects (*i.e.*, no subtyping nor inheritance—*yet*).

[1] ... and pronounced "*hackle*"—not to be confused with an otherwise known programming language of greater notoriety and whose name is the first name of Prof. Haskell B. Curry.

Chapter 8

Conclusion

Appendix A

A word on traceability

A.1 Relating concrete and abstract syntax

Error traceability...

A.1.1 Syntax errors

A.1.2 Static Semantics errors

Typing errors

Other Static Semantics errors

A.1.3 Dynamic Semantics errors

Runtime errors

Java errors

A.2 Displaying and reading

... in concrete/abstract syntax.

A.2.1 Displaying

A.2.2 Reading

A.2.3 Concretizing abstract syntax down

... with writing tables.

A.2.4 Abstracting concrete syntax away

... with reading tables.

Appendix B

A four-panelled architecture

B.1 The Complete Kernel

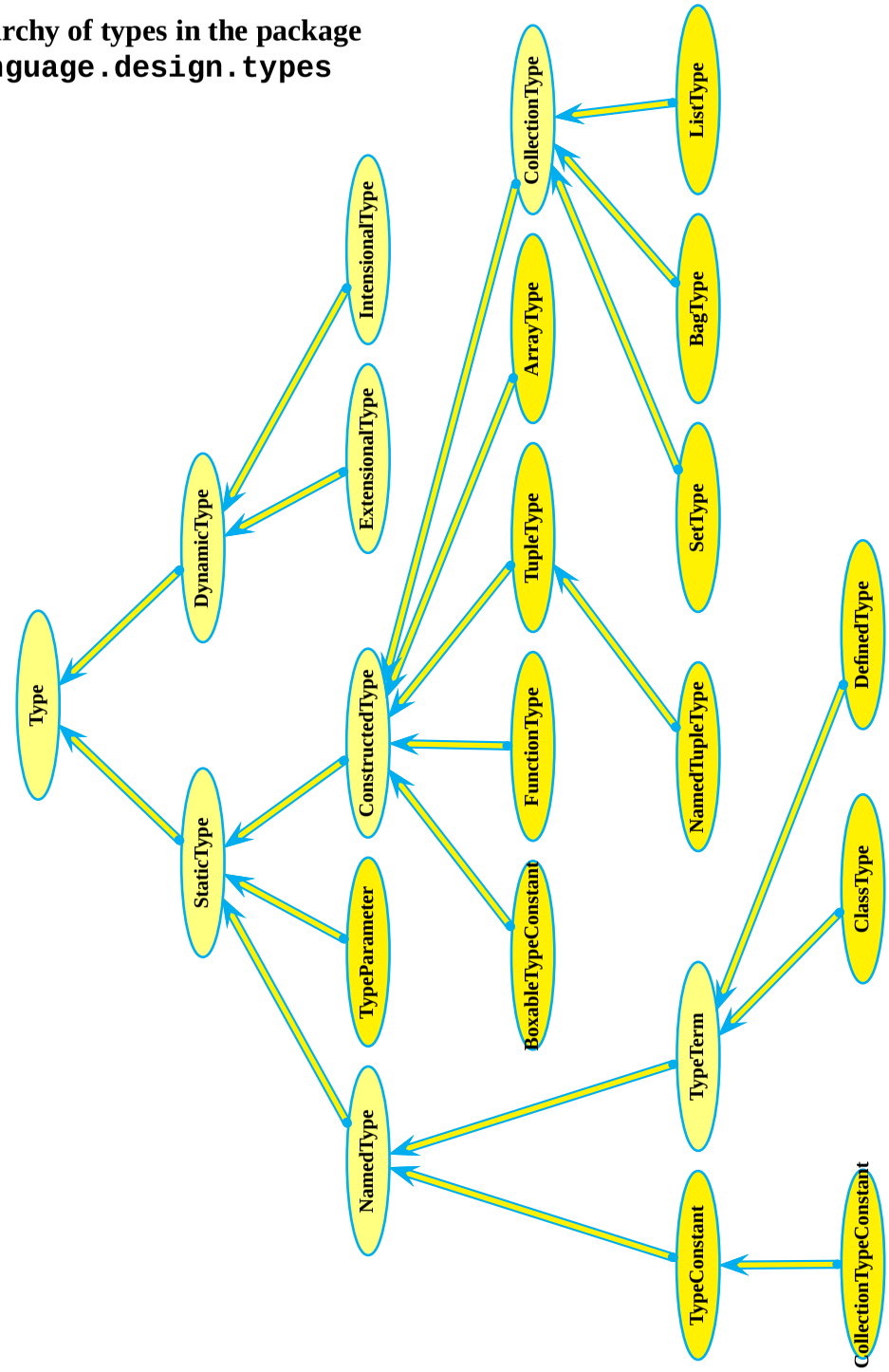
B.1.1 Sanitizing

B.1.2 Type checking *vs.* inference

B.1.3 Compiling

B.2 The Complete Type System

Class hierarchy of types in the package `ilog.language.design.types`



B.2.1 The type prover

B.3 Structure of the `TypeChecker`

An object of the class `ilog.language.design.types.TypeChecker` is a backtracking prover that establishes various kinds of *goals*. The most common goal kind established by a type checker is a *typing goal*—but there are others.

A `TypingGoal` object is a pair consisting of an expression and a type. Proving a typing goal amounts to unifying its expression component's type with its type component. Such goals are spawned by the type checking method of expressions as per their type checking rules. Some globally defined symbols having multiple types, it is necessary to keep choices of these and backtrack to alternative types upon failure. Thus, a `TypeChecker` object maintains all the necessary structures for undoing the effects that happened since the last choice point. These effects are:

1. type variable binding,
2. function type currying,
3. application expression currying.

In addition, it is also necessary to remember all `Goal` objects that were proven since the last choice point in order to prove them anew upon backtracking to an alternative choice. This is necessary because the goals are spawned by calls to the `typeCheck` method of expressions that may be exited long before a failure occurs. Then, all the original typing goals that were spawned in the mean time since the current choice point's goal must be reestablished. In order for this to work, any choice points that were associated to these original goals must also be recovered. To enable this, when a choice point is created for a `Global` symbol, choices are linked in the reverse order (*i.e.*, ending in the original goal) to enable reinstating all choices that were tried for this goal.

In order to coordinate type proving, a typechecker object is passed to all type checking and unification methods as an argument in order to record any effect in the appropriate trail.

To recapitulate, the structures of a `TypeChecker` object are:

- a *goal stack* containing *goal* objects (*e.g.*, `TypingGoal`) that are yet to be proven;
- a *binding trail stack* containing type variables and boxing masks to reset to "unbound" upon backtracking;
- a *function type currying trail* containing 4-tuples of the form (function type, previous domains, previous range, previous boxing mask) for resetting the function type to the recorded domains, range, and mask upon backtracking;

- an *application currying trail* containing triples of the form (application type, previous function, previous arguments) for resetting the application to the recorded function and arguments upon backtracking;
- a *goal trail* containing `TypingGoal` objects that have been proven since the last choice point, and must be reproven upon backtracking;
- a *choice-point stack* whose entries consists of:
  - a queue of `TypingGoalEntry` objects wherefrom to constructs new `TypingGoal` objects to try upon failure;
  - pointers to all trails up to which to undo effects.

B.3.1 The type constructs

B.3.2 Defining new types

B.4 The Basic Instruction Set

B.5 The Complete Backend

B.5.1 The `Runtime` class

B.5.2 The `RuntimeObject` class

B.5.3 The `DisplayManager` class

B.5.4 The `ErrorManager` class

Bibliography

[1] Luca Cardelli. The functional abstract machine. *Polymorphism, the ML/LCF/Hope Newsletter*, I(1), 1983. (Also Technical Report TR-107, AT&T Bell Laboratories, April 1983.).

[2] Hassan Aït-Kaci. An introduction to LIFE—Programming with logic, inheritance, functions, and equations. In Dale Miller, editor, *Proceedings of the Symposium on Logic Programming*. The MIT Press, 1993.

[3] Hassan Aït-Kaci. Jacc—Just another compiler compiler.[1] Optimization Group Technical Report *forthcoming*, ILOG, Gentilly, France, forthcoming 2002.

[4] Peter Landin. The mechanical evaluation of expressions. *Communications of the ACM*, 1964.

[5] Xavier Leroy. Boxing and unboxing in polymorphically typed languages. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL'92)*, 1992.

[6] Pierre Weiss and Xavier Leroy. The CAML compiler. Research report, INRIA, Rocquencourt, France, 1994.

[1] Jacc is a java-based software that generates a `LALR(1)` parsing automaton from a familiar `yacc`-like action-annotated context-free grammar. it provides several useful extensions to `yacc`'s parsing capabilities (*e.g.*, dynamic operator definitions *à la* PROLOG, non-terminal subclassing, *etc.*., ...). Jacc is the property of ILOG but is not part of the software products sold and/or maintained by ILOG—it is not this author's interest to commercialize Jacc (at least not in the immediate future and in its current state), but upon specific request, and on a per-case basis, compiled java classes (not sources) for Jacc may be made available on an *"as is"* basis if it is worth ILOG's and this author's time to do so.