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

June 3, 2002

(INCOMPLETE DRAFT)

This document describes the design of an abstract reusable programming language architect
and its implementation in Java. It represents the basis of the redesign of ILOG's New Gene
tion OPL (hereafter referred to as NGO), and constitutes the second facet of a larger soon-to
proposed ILOG R&D-wide project whose purpose would be to enable the quick integration of r
useful programming abstractions into software at large,[1] insofar as these abstract and reusable c
structs, and any well-typed compositions thereof, may be instantiated in various modular langu
configurations.[2]

---

[1] ILOG's, for one, *intra-* and/or *extra-* company…

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

Contents

Chapter 1

Introduction

This document's purpose is to describe, explain, and justify the design of the `ilog.languag` `.design` package. Its main goal is to serve as a specification as well as a documentation of details of various of its intricacies. As such, it serves mainly its author helping him to keep tr of subtleties he alone may know of but may not remember—at least not in full detail—and, course, it is meant for the sake of the few, the proud, the *"volunteer"* pre-α-testers of the viabi of the whole design—especially the NGO design team, and any others having been exposed, w or nilly, to some of, or the whole package![1]

---

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

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 nar
   and global names, and establish pointers from the local variable occurrences to the abstr
   tion 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
   or several possible ones (only expressions that have a unique unambiguous type are furt
   processed);
3. *“sort-sanitized”*—in the context of a `Sanitizer` to discriminate between those local v
   ables that are of primitive Java types (`int` or `double`) or of `Object` type (this is necess
   because the set-up means to use unboxed values of primitive types for efficiency reaso
   this second “sanitization” phase is also used to compute offsets for local names (*i.e.*,
   called *de Bruijn indices*) for each type sort;
4. *compiled*—in the context of a `Compiler` to generate the sequence of instructions wh
   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 r 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 prototypes of the methods that must be implemented by all expression subclasses. The subclas 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;
- `Global`: global name;
- `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;
- `Definition`: definition of a global name with an expression defining it in a global sto
- `Parameter`: a function’s formal parameter (really a pseudo-expression as it is not fu processed as a real expression and is used as a shared type information repository for occurrences in a function’s body of the variable it stands for);
- `Assignment`: construct to set the value of a `local` or a `global` variable;
- `NewObject`: construct to create a new object;
- `FieldUpdate`: construct to update the value of an object’s field;
- `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;
- `Dummy`: temporary place holder in lieu of a name prior to being discriminated into a local or global one.
- `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 rule* is a formula of the form:

$$\frac{J_1, \ldots, J_n}{J} \tag{3.1}$$

---

where J and the J_i's, $i = 0, \ldots, n,\ n \geq 0$, are *typing judgments*. When $n = 0$, the rule is cal an *axiom* and is written with an empty "numerator."

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

$$\frac{J_1, \ldots, J_n}{J} \qquad \texttt{if} \quad c(J_1, \ldots, J_n) \qquad (3$$

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

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

A typing rule, or its (un/conditional) typing axiom form, is best read backwards (or upwards)— from the rule's *conclusion* (the bottom part, or "denominator") to the rule's *premises* (the top p or "numerator"). Namely, the rule of the form:

$$\frac{\Gamma_1 \ \vdash \ e_1 \ : \ T_1, \ \ldots, \ \Gamma_n \ \vdash \ e_n \ : \ T_n}{\Gamma \ \vdash \ e \ : \ T} \qquad (3$$

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`** *. . . , the expression e_n has type T_n under typing context Γ_n."*

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

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

$$\Gamma[x : T](y) \ \stackrel{\text{def}}{=} \ \begin{cases} T & \text{if } y = x; \\ \Gamma(x) & \text{otherwise.} \end{cases} \qquad (3$$

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

$$\frac{\Gamma_1 \ \vdash \ e_1 \ : \ T_1, \ \ldots, \ \Gamma_n \ \vdash \ e_n \ : \ T_n}{\Gamma \ \vdash \ e \ : \ T} \qquad \texttt{if} \quad c(\Gamma, \Gamma_1, \ldots, \Gamma_n, e, e_1, \ldots, e_n, T, T_1, \ldots, T_n) \ \ (3$$

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 c texts, 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 *(unconditional) typing axiom*:

$$\frac{}{\Gamma \vdash e : T} \tag{3.6}$$

is read thus:

> *"The expression e has type T under typing context Γ."*

The *(conditional) typing axiom* form:

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

where $c(\Gamma, e, T)$ is a boolean meta-condition on typing context Γ, expression e, and type T, is read thus:

> *"`If` the meta-condition $c(\Gamma, e, T)$ holds `then` the expression e has type T under typing context Γ."*

For example,

$$\frac{\Gamma \vdash c : \mathfrak{Boolean},\ \Gamma \vdash e_1 : T,\ \ \Gamma \vdash e_2 : T}{\Gamma \vdash \mathfrak{if}\ c\ \mathfrak{then}\ e_1\ \mathfrak{else}\ e_2 : T} \tag{3.8}$$

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."*

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:

$$\mathfrak{compile}[\![e]\!] = \begin{array}{l} \texttt{INSTRUCTION}_1 \\ \vdots \\ \texttt{INSTRUCTION}_n \end{array} \tag{3.9}$$

3.3.1 Constant

ABSTRACT SYNTAX

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

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

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

TYPING RULE

The typing rules for each kind of constant are:

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

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

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

$$\begin{array}{lll} \texttt{[int]} & \dfrac{}{\Gamma \vdash n : \mathfrak{Int}} & \texttt{if} \quad n \text{ is an integer} \\ \texttt{[real]} & \dfrac{}{\Gamma \vdash n : \mathfrak{Real}} & \texttt{if} \quad n \text{ is a floating-point number} \\ \texttt{[char]} & \dfrac{}{\Gamma \vdash c : \mathfrak{Char}} & \texttt{if} \quad c \text{ is a character} \\ \texttt{[string]} & \dfrac{}{\Gamma \vdash s : \mathfrak{String}} & \texttt{if} \quad s \text{ is a string} \end{array} \tag{3.10}$$

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

3.3.2 Abstraction

ABSTRACT SYNTAX

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

TYPING RULE

$$\frac{\Gamma[x_1 : T_1] \cdots [x_n : T_n] \vdash e : T}{\Gamma \vdash \mathfrak{function}\ x_1, \ldots, x_n \cdot e : T_1, \ldots, T_n \rightarrow T} \tag{3.11}$$

3.3.3 Application

ABSTRACT SYNTAX

$$f(e_1, \ldots, e_n)$$

TYPING RULE

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

(3.

3.3.4 Local

3.3.5 Global

3.3.6 IfThenElse

ABSTRACT SYNTAX

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

OPERATIONAL SEMANTICS

TYPING RULE

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

(3.

COMPILING RULE

$$\begin{array}{lrl} \mathfrak{compile}[\![\mathfrak{if}\ c\ \mathfrak{then}\ e_1\ \mathfrak{else}\ e_2]\!] = & & \mathfrak{compile}[\![c]\!] \\ & & \textsc{Jump\_On\_False}\ \texttt{jof} \\ & & \mathfrak{compile}[\![e_1]\!] \\ & & \textsc{Jump}\ \texttt{jmp} \\ & \texttt{jof}: & \mathfrak{compile}[\![e_2]\!] \\ & \texttt{jmp}: & \ldots \end{array}$$

(3.

3.3.7 AndOr

ABSTRACT SYNTAX

$$e_1 \ \mathfrak{and}/\mathfrak{or} \ e_2$$

TYPING RULE

$$\frac{\Gamma \vdash e_1 : \mathfrak{Boolean}, \quad \Gamma \vdash e_2 : \mathfrak{Boolean}}{\Gamma \vdash e_1 \ \mathfrak{and}/\mathfrak{or} \ e_2 : \mathfrak{Boolean}} \tag{3.15}$$

And

COMPILING RULE

$$\begin{array}{rrl} \mathfrak{compile}[\![e_1 \ \mathfrak{and} \ e_2]\!] = & & \mathfrak{compile}[\![e_1]\!] \\ & & \textsc{Jump\_On\_False} \ \mathtt{jof} \\ & & \mathfrak{compile}[\![e_2]\!] \\ & & \textsc{Jump\_On\_True} \ \mathtt{jot} \\ & \mathtt{jof}: & \textsc{Push\_False} \\ & & \textsc{Jump} \ \mathtt{jmp} \\ & \mathtt{jot}: & \textsc{Push\_True} \\ & \mathtt{jmp}: & \ldots \end{array} \tag{3.16}$$

Or

COMPILING RULE

$$\begin{array}{rrl} \mathfrak{compile}[\![e_1 \ \mathfrak{or} \ e_2]\!] = & & \mathfrak{compile}[\![e_1]\!] \\ & & \textsc{Jump\_On\_True} \ \mathtt{jot} \\ & & \mathfrak{compile}[\![e_2]\!] \\ & & \textsc{Jump\_On\_False} \ \mathtt{jof} \\ & \mathtt{jot}: & \textsc{Push\_True} \\ & & \textsc{Jump} \ \mathtt{jmp} \\ & \mathtt{jof}: & \textsc{Push\_False} \\ & \mathtt{jmp}: & \ldots \end{array} \tag{3.17}$$

3.3.8 Sequence

ABSTRACT SYNTAX

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

TYPING RULE

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

(3.

COMPILING RULE

$$\begin{array}{rl} \mathfrak{compile}[\![\{ \ e_1; \ \ldots ; e_n \ \}]\!] = & \mathfrak{compile}[\![e_1]\!] \\ & \textsc{Pop\_}\mathit{sort}(e_1) \\ & \vdots \\ & \mathfrak{compile}[\![e_n]\!] \end{array}$$

(3.

3.3.9 Let

3.3.10 Loop

ABSTRACT SYNTAX

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

(3.

OPERATIONAL SEMANTICS

TYPING RULE

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

(3.

COMPILING RULE

$$
\mathrm{compile}[\![\mathfrak{while}\ c\ \mathfrak{do}\ e]\!] = \begin{array}{rl} \texttt{loop}: & \mathrm{compile}[\![c]\!] \\ & \text{JUMP\_ON\_FALSE}\ \texttt{jof} \\ & \mathrm{compile}[\![e]\!] \\ & \text{JUMP}\ \texttt{loop} \\ \texttt{jof}: & \text{PUSH\_VOID} \end{array} \tag{3.22}
$$

3.3.11 ExitWithValue

ABSTRACT SYNTAX

$\mathfrak{exit\ with}\ v$

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; it

1. pops the result from its result stack;[5]
2. restores the (previously saved) runtime state;
3. 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 may *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 $\mathfrak{exit\ with}\ v$ kernel construction $\mathfrak{exit\ with}\ v$ 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.

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.

---

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

First of all, from its very name $\mathfrak{exit\ with}\ v$ assumes that computation has *entered* that from whic
must *exit*. This is an *exitable* abstraction; that is, the latest λ-abstraction having the property of
ing *exitable*. Not all abstractions are exitable. For example, any abstraction that is generated as
of the target of some other kernel expression's syntacting sugar (*e.g.*, $\mathfrak{let}\ x_1 = e_1; \ldots; x_n = e_n;$
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
stractions 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 ret
type of the abstraction being exited is. In other words:

$$
\frac{\Gamma \vdash \aleph_\Gamma : T' \rightarrow T, \quad \Gamma \vdash v : T}{\Gamma \vdash \mathfrak{exit\ with}\ v : T}
$$

(3.

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
value of $\aleph_\Gamma$, the type in Γ of the latest exitable abstraction, if any such exists; (if none d
a static semantics error is triggered to indicate that it is impossible to exit from anywh
before first entering somewhere).

2. Congruently, the PUSH_CLOSURE instruction must take care of chaining the state it pus
in the saved state stack of the runtime system each time a closure coming from an exita
abstraction is entered; (dually, this exitable state stack must also be popped upon "fall
off"—*i.e.*, normally exiting—an exitable closure).

3. New NL_RETURN instructions (for each runtime sort) must be defined like their correspond
RETURN instructions except that the runtime state to restore is the one popped out of
exitable state stack.

COMPILING RULE

$$
\mathrm{compile}[\![\mathfrak{exit\ with}\ v]\!] = \begin{array}{l} \mathrm{compile}[\![v]\!] \\ \text{NL\_RETURN\_}\mathit{sort}(v) \end{array}
$$

(3.

3.3.12 Definition

3.3.13 Parameter

3.3.14 Assignment

3.3.15 NewObject

3.3.16 FieldUpdate

3.3.17 NewArray

3.3.18 ArraySlot

3.3.19 ArraySlotUpdate

3.3.20 Tuple

3.3.21 NamedTuple

3.3.22 TupleProjection

3.3.23 TupleUpdate

3.3.24 Dummy

3.3.25 ArrayExtension

3.3.26 ArrayInitializer

3.3.27 Homomorphism

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:

$$
\begin{aligned}
\mathbf{hom}_{\oplus}^{\mathbb{1}_{\oplus}}(f)[\,] &= \mathbb{1}_{\oplus} \\
\mathbf{hom}_{\oplus}^{\mathbb{1}_{\oplus}}(f)[H|T] &= f(H) \oplus \mathbf{hom}_{\oplus}^{\mathbb{1}_{\oplus}}(f)T
\end{aligned}
\qquad (3.
$$

Clearly, this scheme extends a function f to a homomorphism of monoids, from the monoid of l 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 c function (f) to an element of collection monoid (*i.e.*, a data structure such as a set, a list, c bag), the image monoid being implicitly defined by the binary operation ($\oplus$)—also called *accumulation* operation. It is made to work iteratively.

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

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

Therefore, a *collection homomorphism* expression constructing a collection of type $coll(T)$ c sists 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 $\mathfrak{function}\ x\ \cdot\ x \oplus \mathbb{1}_{\oplus}$ (*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:

$$\langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid q_1, \ldots, q_n \} \tag{3.26}$$

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 a *boolean* expression 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:

$$\begin{array}{lcl}
\langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid \} & \overset{\text{def}}{=} & e \oplus \mathbb{1}_{\oplus} \\
\langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid x\ \leftarrow\ e', Q \} & \overset{\text{def}}{=} & \mathfrak{hom}_{\oplus}^{\mathbb{1}_{\oplus}}[\lambda x. \langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid Q \}](e') \\
\langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid c, Q \} & \overset{\text{def}}{=} & \mathfrak{if}\ c\ \mathfrak{then}\ \langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid Q \}\ \mathfrak{else}\ \mathbb{1}_{\oplus}
\end{array} \tag{3.27}$$

This is explained more formally in Section 3.3.28.

Comprehensions are also interesting as they may be subject to transformations leading to m
efficient evaluation than their simple “nested loops” operational semantics (by using “unnesti
techniques and using relational operations as implementation instructions). At any rate, homom
phisms are here treated “naively” and compiled as simple loops.

3.3.28 Comprehension

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

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

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

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

$$\begin{array}{lcl}
\oplus\{e \parallel \} & \overset{\mathit{def}}{=} & \begin{cases} \mathfrak{u}_{\oplus}(e) & \textit{if } \oplus \textit{ is a collection monoid} \\ e & \textit{if } \oplus \textit{ is a primitive monoid} \end{cases} \\
\oplus\{e \parallel x\ \leftarrow\ e', Q\} & \overset{\mathit{def}}{=} & \mathfrak{hom}_{\odot}^{\oplus}[\lambda x.\ \oplus \{e \parallel Q\}](e') \\
\oplus\{e \parallel c, Q\} & \overset{\mathit{def}}{=} & \mathfrak{if}\ c\ \mathfrak{then}\ \oplus \{e \parallel Q\}\ \mathfrak{else}\ \mathfrak{z}_{\oplus}
\end{array}$$

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 (3.25)), 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.

3.3.29 CompiledExpression

Chapter 4

The Type System

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 differences that will be explained along the way—in other words, types presented with a gramm such as:

[1] *Type* $::=$ *SimpleType* $|$ *Polytype*

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

[3] *BasicType* $::= \mathfrak{Int} \mid \mathfrak{Real} \mid \mathfrak{Boolean} \mid \ldots$

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

[5] *TypeParameter* $::= \alpha \mid \alpha' \mid \ldots \mid \beta \mid \beta' \mid \ldots$

[6] *PolyType* $::= \forall$ *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 d ensures decidability of type inference. BTW, the 2*nd* order comes from the fact that the quantifier applies to 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 applying Equation (4.1) as a rewrite rule, although *backwards* (*i.e.*, from right to left) and as much as possible. The second (renaming) phase consists in making a consistent copy of all types reachable from a type's root.

---

[2] For the reader who might wonder what all this has to do with Indian cooking: it does not. It comes from Prof. Haskell B. Curry's last name. Curry was one of the two mathematicians/logicians (along with ?. Feys) who conceived *Combinator Logic* and *Combinator Calculus*, and made extensive use of the isomorphism of Equation (4.1)—hence the folklore's coining of the verb *to curry—(currying, curryed),—* in French: *curryfier—(curryfication, curryfié)*. The homonymy is often amusingly mistaken for an exotic way of [un]spicing functions.

4.1.5 Copying

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

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

4.1.7 Unifying

Unifying two types is the operation of filling in missing information (*i.e.*, type parameters) in e
with existing information from the other by side-effecting (*i.e.*, binding) the missing informat
(*i.e.*, the type parameters) to point to the part of the existing information from the other type t
should be equal to (*i.e.*, their values). Note that, like logical variables in Logic Programming, t
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 p
morphic type must work for all instantiations of this type's type parameters into either primi
unboxed types (*e.g.*, $\mathfrak{Int}$, $\mathfrak{Real}$, *etc.*) or boxed types. The problem this poses is: how can
compile a polymorphic function into code that would correctly know what the actual runtime s
of the function's runtime arguments and returned value are, *before the function type is actu
instantiated into a (possibly monomorphic) type*?[3] The problem was addressed by Xavier Le
10 years ago [5] and he proposed a solution.[4] Leroy's method is based on the use of type an

---

[3] Besides compiling distinct copies for all possible runtime sort instantiations (like, *e.g.*, C++ template functi
nor recompiling each time a specific instantiation is needed. The former is not acceptable because its tends to in
the code space explosively. The latter can neither be envisaged because it goes against a few (rightfully) sacros
principles like separate compilation and abstract library interfacing—imaging having to recompile code from a lib
everytime you want to use it!

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

tation 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 garanteed 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):
    - Set extension type;
    - Int range extension type (close interval of ints);

---

[5] See Appendix Section B.2 on Page 52 for the complete class hierarchy of types in the package `ilog.language.design.types`.

    - Real range extension type (close interval of reals).

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

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

4.3 Type Definitions

Type definitions are provided both for convenience of making programs more legible by giv
“logical” names (or terms) to otherwise verbose types, and that of hiding information details
a type making it act as a new type altogether. The former facility is that of providing *aliases*
types (exactly like a preprocessor’s macros get expanded right away into their textual equivalen
while the latter offers the convenience of defining *new* types in terms of existing ones, but hid
this information. It follows from this distinction that a type alias is *always* structurally equival
to its value (in fact an alias disappears as soon as it is read in, being parsed away into the struct
defining it). By contrast, a defined type is *never* structurally equivalent to its value nor any ot
type—it is only equivalent to itself. To enable meaningful computation with a defined type, t
meta-(de/con)structors are thus provided: one for explicitly *casting* a defined type into the t
that defines it, and one explicitly seeing a type as a specified defined type (if such a defined t
does exist and with this type as definition).

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

The `typeTable` variable contains the naming table for types and the `symbolTable` varia
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 mana
symbols: `ilog.language.design.types.Tables`. The names can be those of types
values. They are *global* names.[6] The type namespace is independent of the value namespace—
the same name can denote a value and a type.

---

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

4.3.1 Type aliasing

4.3.2 Type hiding

4.4 Static types

The static type system...

4.4.1 Primitive types

Boxable types

- Void
- Int
- Real
- Char
- Boolean

Boxed types

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

4.4.2 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

Set types

Class types

4.4.3 Polymorphic types

4.4.4 Type aliasing

4.4.5 Type hiding

4.5 Dynamic types

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

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

This will ensure that the actual value places in the slot expecting a certain type does respects ac tionnal constraints that may only be verified with some runtime values. Generally, dynamic ty are so-called *dependent* types (such as, *e.g.*, `array_of_size`(n), a “safe” array type depend 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 meth `public boolean verifyCondition()` that is invoked systematically by code genera for dynamically typed function parameters and for locations that are the target of updates (array slot update, object field update, tuple field update) at compilation of abstractions and vari 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 OPL, and hence NGO, types). Namely,

- extensional types;

- non-negative numbers—or more generally, Boolean-assertion types (of which non-negative number types are instances).

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).

The notion of extensional type

Set types

Int range types

Float range types

Enum types

4.5.1 Conditional types

Non-negative numbers

4.5.2 The notion of dynamically constrained type (`int+`, `float+`,...)

The notion of boolean-assserted type

4.5.3 Extensional types

Chapter 5

The instruction base

The complete list of instructions that are currently defined is:

1. **Do-nothing instruction:**
    (a) No_Op
2. **Push instructions:**
    (a) Push_I
    (b) Push_O
    (c) Push_R
    (d) Push_Offset_I
    (e) Push_Offset_O
    (f) Push_Offset_R
    (g) Push_Tuple
    (h) Push_Set_I
    (i) Push_Set_R
    (j) Push_Set_O
    (k) Push_Int_Rng
    (l) Push_Real_Rng
    (m) Push_Closure
    (n) Push_New_Object
3. **Subroutine instructions:**
    (a) Apply
    (b) Apply_Hom_I
    (c) Apply_Hom_R
    (d) Apply_Hom_O
    (e) Apply_Ip_Hom_I
    (f) Apply_Ip_Hom_R
    (g) Apply_Ip_Hom_O
    (h) Apply_Coll_I
    (i) Apply_Coll_R
    (j) Apply_Coll_O
    (k) Apply_Coll_Hom_I
    (l) Apply_Coll_Hom_R
    (m) Apply_Coll_Hom_O
    (n) Apply_Ip_Coll_Hom_I
    (o) Apply_Ip_Coll_Hom_R
    (p) Apply_Ip_Coll_Hom_O
    (q) Call
    (r) End
    (s) Return_I
    (t) Return_R
    (u) Return_O
    (v) NL_Return_I
    (w) NL_Return_R
    (x) NL_Return_O
4. **Pop instructions:**
    (a) Pop_I

(b) POP_O
(c) POP_R

5. **Relocatable instructions:**

(a) JUMP
(b) JUMP_ON_FALSE
(c) JUMP_ON_TRUE

6. **Conversion instructions:**

(a) I_TO_O
(b) I_TO_R
(c) O_TO_I
(d) O_TO_R
(e) R_TO_I
(f) R_TO_O
(g) ARRAY_TO_MAP_I
(h) ARRAY_TO_MAP_R
(i) ARRAY_TO_MAP_O
(j) MAP_TO_ARRAY_O
(k) CHECK_ARRAY_SIZE
(l) RECONCILE_INDEXABLES
(m) ARRAY_INITIALIZE
(n) SHUFFLE_MAP_I
(o) SHUFFLE_MAP_R
(p) SHUFFLE_MAP_O

7. **Assignment instructions:**

(a) SET_GLOBAL
(b) SET_OFFSET_I
(c) SET_OFFSET_O
(d) SET_OFFSET_R

8. **Tuple component instructions:**

(a) GET_TUPLE_I
(b) GET_TUPLE_R
(c) GET_TUPLE_O
(d) SET_TUPLE_I
(e) SET_TUPLE_R
(f) SET_TUPLE_O

9. **Array/Map allocation instructions:**

(a) PUSH_ARRAY_I
(b) PUSH_ARRAY_R
(c) PUSH_ARRAY_O
(d) PUSH_MAP_I
(e) PUSH_MAP_R
(f) PUSH_MAP_O
(g) MAKE_ARRAY_I
(h) MAKE_ARRAY_R
(i) MAKE_ARRAY_O
(j) MAKE_MAP_I
(k) MAKE_MAP_R
(l) MAKE_MAP_O
(m) FILL_ARRAY_IA
(n) FILL_ARRAY_IM
(o) FILL_ARRAY_OA
(p) FILL_ARRAY_OM
(q) FILL_ARRAY_RA
(r) FILL_ARRAY_RM
(s) FILL_MAP_IA
(t) FILL_MAP_IM
(u) FILL_MAP_OA
(v) FILL_MAP_OM
(w) FILL_MAP_RA
(x) FILL_MAP_RM

10. **Array/Map slot instructions:**

(a) GET_ARRAY_I
(b) GET_INT_INDEXED_MAP_I
(c) GET_INT_INDEXED_MAP_O

(d) GET_INT_INDEXED_MAP_R
(e) GET_MAP_I
(f) GET_ARRAY_O
(g) GET_MAP_O
(h) GET_ARRAY_R
(i) GET_MAP_R
(j) SET_ARRAY_I
(k) SET_INT_INDEXED_MAP_I
(l) SET_INT_INDEXED_MAP_O
(m) SET_INT_INDEXED_MAP_R
(n) SET_MAP_I
(o) SET_ARRAY_O
(p) SET_MAP_O
(q) SET_ARRAY_R
(r) SET_MAP_R

11. **Field instructions:**

(a) GET_FIELD_I
(b) GET_FIELD_O
(c) GET_FIELD_R
(d) SET_FIELD_I
(e) SET_FIELD_O
(f) SET_FIELD_R

12. **Built-in operations:**

(a) **Arithmetic operations:**

i. ADD_II
ii. ADD_IR
iii. ADD_RI
iv. ADD_RR
v. SUB_II
vi. SUB_IR
vii. SUB_RI
viii. SUB_RR
ix. MINUS_I
x. MINUS_R
xi. MUL_II
xii. MUL_IR
xiii. MUL_RI
xiv. MUL_RR
xv. DIV_II
xvi. DIV_IR
xvii. DIV_RI
xviii. DIV_RR
xix. MODULUS
xx. MIN_II
xxi. MIN_IR
xxii. MIN_RI
xxiii. MIN_RR
xxiv. MAX_II
xxv. MAX_IR
xxvi. MAX_RI
xxvii. MAX_RR
xxviii. ABS_I_RI
xxix. ABS_R
xxx. SQRT
xxxi. POWER

(b) **Arithmetic relations:**

i. EQU_II
ii. EQU_OO
iii. EQU_RR
iv. NEQ_II
v. NEQ_OO
vi. NEQ_RR
vii. GTE_II
viii. GTE_IR
ix. GTE_RI
x. GTE_RR
xi. GRT_II
xii. GRT_IR
xiii. GRT_RI
xiv. GRT_RR
xv. LTE_II
xvi. LTE_IR
xvii. LTE_RI

    xviii. LTE_RR
    xix. LST_II
    xx. LST_IR
    xxi. LST_RI
    xxii. LST_RR
  (c) **Boolean operations:**
    i. NOT
  (d) **Map and Size operations:**
    i. MAP_SIZE
    ii. ARRAY_SIZE
    iii. INDEXABLE_SIZE
    iv. GET_INDEXABLE
  (e) **Container operations:**
    i. BELONGS_I
    ii. BELONGS_O
    iii. BELONGS_R
  (f) **Set operations:**
    i. SET_COPY
    ii. MAKE_SET_I
    iii. MAKE_SET_O
    iv. MAKE_SET_R
    v. SET_DIFF
    vi. SET_SYM_DIFF
    vii. INTER
    viii. UNION
    ix. D_SET_DIFF
    x. D_SET_SYM_DIFF
    xi. D_INTER
    xii. D_UNION
  (g) **Set relations:**
    i. SUBSET
  (h) **Set element operations:**
    i. SET_ADD_I
    ii. SET_ADD_R
    iii. SET_ADD_O
    iv. SET_RMV_I
    v. SET_RMV_R
    vi. SET_RMV_O
    vii. FIRST_I
    viii. FIRST_O
    ix. FIRST_R
    x. LAST_I
    xi. LAST_O
    xii. LAST_R
    xiii. NEXT_I
    xiv. NEXT_C_I
    xv. NEXT_O
    xvi. NEXT_C_O
    xvii. NEXT_R
    xviii. NEXT_C_R
    xix. ORD_I
    xx. ORD_O
    xxi. ORD_R
    xxii. PREV_I
    xxiii. PREV_C_I
    xxiv. PREV_O
    xxv. PREV_C_O
    xxvi. PREV_R
    xxvii. PREV_C_R
  (i) **Range operations:**
    i. INT_RNG_UB
    ii. INT_RNG_LB
    iii. REAL_RNG_UB
    iv. REAL_RNG_LB
  (j) **String operations:**
    i. STRCON
  (k) **I/O operations:**
    i. WRITE_I
    ii. WRITE_O
    iii. WRITE_R

13. **Dummy instructions:**

  (a) DUMMY_EQU

  (b) DUMMY_NEQ
  (c) DUMMY_AND
  (d) DUMMY_OR
  (e) DUMMY_STRCON
  (f) DUMMY_WRITE
  (g) DUMMY_SIZE
  (h) DUMMY_SET_ADD
  (i) DUMMY_SET_RMV
  (j) DUMMY_BELONGS
  (k) DUMMY_ORD
  (l) DUMMY_FIRST
  (m) DUMMY_LAST
  (n) DUMMY_NEXT
  (o) DUMMY_NEXT_C
  (p) DUMMY_PREV
  (q) 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 envir
ment context shared by `Instruction`s being executed. It encapsulates a state of comptutat
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 computati
and methods that are used by instructions to effect this state as they are executed. Thus, e
instruction class defines an `execute(Runtime)` method that specifies its operational seman
as a state transformation of its given runtime context.

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

Runtime states encapsulated by objects in this class are essentially those of a stack automat
specifically conceived to support the computations of a higher-order functional language with le
cal closures - *i.e.*, a λ-calculus machine - extended to support additional features - *e.g.*, assignm
side-effects, objects, automatic currying... As such it may viewed as an optimized variant of Pe
Landin's SECD machine [4]—in the same spirit as Luca Cardelli's Functional Abstract Mach
(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
being confined to boxed values for primitive type computations, three concurrent sets of structu

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_L
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 demonstr
our architecture: the expressive power of the kernel language and the workings of its type-checl
compiler, and runtime systems. It is an imperative functional language with objects, where fu
tions are first-class citizens. HAK_LL has a surface syntax for an interactive language that
define top-level constructs and evaluate expressions. It supports 2nd-order (ML-like) type po
morphism, automatic currying, multiple type overloading, dynamic operator overloading, as v
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 gre
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

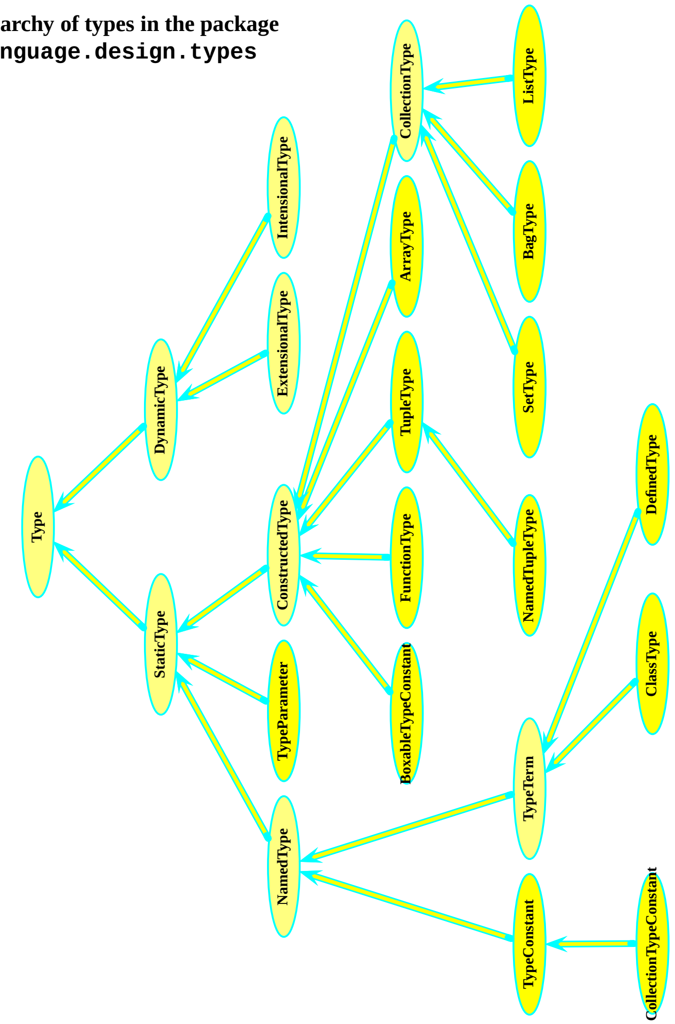
B.1.1 Sanitizing

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 backtrack prover that establishes various kinds of *goals*. The most common goal kind established by a t checker is a *typing goal*—but there are others.[1] A `TypingGoal` object is a pair consisting of expression and a type. Proving a typing goal amounts to unifying its expression component's t with its type component. Such goals are spawned by the type checking method of expressions per their type checking rules. Some globally defined symbols having multiple types, it is necess to keep choices of these and backtrack to alternative types upon failure. Thus, a `TypeCheck` object maintains all the necessary structures for undoing the effects that happened since the 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 cho point in order to prove them anew upon backtracking to an alternative choice. This is necess because the goals are spawned by calls to the `typeCheck` method of expressions that may exited long before a failure occurs. Then, all the original typing goals that were spawned in mean time since the current choice point's goal must be reestablished. In order for this to wo any choice points that were associated to these original goals must also be recovered. To ena this, when a choice point is created for a `Global` symbol, choices are linked in the reverse or (*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 un cation 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;

---

[1] At the moment, the handled goals are:

- typing goal: $e : T$;
- type unification goal: $T = T'$;
- base type unification goal: $T = base(T')$;

Others are expected (and will!) be introduced, *e.g.*, when we support subtyping constraints: $e <: T$, $T \leq T'$, *etc.*

- 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 Newslet* 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, functic and equations. In Dale Miller, editor, *Proceedings of the Symposium on Logic Programmi* The MIT Press, 1993.

[3] Hassan Aït-Kaci. 𝔍𝔞𝔠𝔠—Just another compiler compiler.[2] Optimization Group Techni Report *forthcoming*, ILOG, Gentilly, France, forthcoming 2002.

[4] Peter Landin. The mechanical evaluation of expressions. *Communications of the ACM*, 19

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

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

---

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