An Abstract, Reusable, and Extensible Programming Language Design Architecture*

Hassan Aït-Kaci

Université Claude Bernard Lyon 1
Villeurbanne, France

hak@acm.org

Abstract. There are a few basic computational concepts that are at the core of all programming languages. The exact elements making out such a set of concepts determine (1) the specific nature of the computational services such a language is designed for, (2) for what users it is intended, and (3) on what devices and in what environment it is to be used. It is therefore possible to propose a set of basic building blocks and operations thereon as combination procedures to enable programming software by specifying desired tasks using a tool-box of generic constructs and meta-operations. Syntax specified through LALR(k) grammar technology can be enhanced with greater recognizing power thanks to a simple augmentation of `yacc` technology. Upon this basis, a set of implementable formal operational semantics constructs may be simply designed and generated (syntax and semantics) *à la carte*, by simple combination of its desired features. The work presented here, and the tools derived from it, may be viewed as a tool box for generating language implementations with a desired set of features. It eases the automatic practical generation of programming language pioneered by Peter Landin's SECD Machine. What is overviewed constitutes a practical computational algebra extending the polymorphically typed λ-Calculus with object/classes and monoid comprehensions. This paper describes a few of the most salient parts of such a system, stressing most specifically any innovative features—formal syntax and semantics. It may be viewed as a high-level tour of a few reusable programming language design techniques prototyped in the form of a set of composable abstract machine constructs and operations.[1]

Keywords: *Programming Language Design; Object-Oriented Programming; Denotational Semantics; Operational Semantics; λ-Calculus; Polymorphic Types; Static/Dynamic Type Checking/Inference; Declarative Collections; Monoid Comprehensions; Intermediate Language; Abstract Machines.*

This article is dedicated to Peter Buneman, a teacher and a friend—for sharing the fun! With fond memories of our Penn days and those Friday afternoon seminars in his office . . .

* Thanks to Val Tannen for his patience, Nabil Layaïda for his comments, and the anonymous referee for catching many glitches and giving good advice in general.

[1] Some of this material was presented as part of the author's keynote address at LDTA 2003 [5]. The source code is available on `https://github.com/ha-k/hak-lang-syntax`.

> The languages people use to communicate with computers differ in their intended aptitudes towards either a particular application area, or in a particular phase of computer use (high level programming, program assembly, job scheduling, *etc.*, ...). They also differ in physical appearance, and more important, in logical structure. The question arises, do the idiosyncrasies reflect basic logical properties of the situations that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments? This question is clearly important if we are trying to predict or influence language evolution.
>
> To answer it we must think in terms, not of languages, but of families of languages. That is to say we must systematize their design so that a new language is a point chosen from a well-mapped space, rather than a laboriously devised construction.
>
> PETER J. LANDIN—"The Next 700 Programming Languages" [31]

1 Introduction

1.1 Motivation—programming language design?

Today, programming languages are designed more formally than they used to be fifty years ago. This is thanks to linguistic research that has led to syntactic science (begetting parser technology) and research in the formal semantics of programming constructs (begetting compiler technology—semantics-preserving translation from human-usable surface syntax to low-level instruction-based machine language). As in the case of a natural language, a grammar is used to control the formation of sentences (programs) that will be understood (interpreted/executed) according to the language's intended (denotational/operational) semantics. Design based on formal syntax and semantics can thus be made operational.

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. To this, we must add the potentially high design costs investing in defining and implementing a new language. These costs affect not only time and effort of design and development, but also the quality of the end product—*viz.*, performance and reliability of the language being designed, not to mention how to justify, let alone guarantee, the correctness of the design's implementation [37].

Fortunately, there have been design tools to help in the process. So-called meta-compilers have been used to great benefit to systematize the design and guarantee a higher quality of language implementation. The "*meta*" part actually applies to 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 an attributed tree language with some *ad hoc* attribute co-dependence interpretation. Even rarer are language design systems that propose abstract and reusable components in the form of expressions of a formal typed

kernel calculus. It is such a system that this work proposes; it gives an essential overview of its design principle and the sort of services it has been designed to render.

This document describes the design of an abstract, reusable, and extensible, programming language architecture and its implementation in Java. What is described represents a generic basis insofar as these abstract and reusable constructs, and any well-typed compositions thereof, may be instantiated in various modular language configurations. It also offers a practical discipline for extending the framework with additional building blocks for new language features as per need. The first facet was the elaboration of Jacc,[2] an advanced system for syntax-directed compiler generation extending `yacc` technology [28].[3] A second facet was the design of a well-typed set of abstract-machine constructs complete enough to represent higher-order functional programming in the form of an object-oriented λ-Calculus, extended with *monoid comprehensions* [14,15,13,22]. A third facet could be the integration of logic-relational (from Logic Programming) and object-relational (from Database Programming) enabling LIFE-technology [4,7] and/or any other $\mathcal{CP}/\mathcal{LP}$ technology to cohabit.

What is described here is therefore a metadesign: it is the design of a design tool. The novelty of what is described here is both in the lexical/syntactical phase and in the typing/execution semantic phase.

The lexical and syntactic phases are innovative in many respects. In particular, they are conservative extensions considerably enhancing the conventional `lex/yacc` technology (or, similarly, `flex/bison`) meta-lexico-syntactical tools [28,19] with more efficient implementation algorithms [34] and recognizing power (*viz.*, overloaded grammar symbols, dynamic operator properties *à la* Prolog. This essentially gives Jacc the recognizing power of LALR(k) grammars, for any $k \geq 1$.) Sections 2.1 and 2.2 give more details on that part of the system.

The interpretation is essentially the same approach as the one advocated by Landin for his Store-Environment-Code-Dump (SECD) machine [30] and optimized by Luca Cardelli in his Functional Abstract Machine (FAM) [16].[4] The abstract machine we present here is but a systematic taking advantage of Java's object-oriented tool-set to put together a modular and extensible set of building blocks for language design. It is sufficiently powerful for expressing higher-order polymorphic object-oriented functional and/or imperative programming languages. This includes declarative collection-processing based on the concept of Monoid Comprehensions as used in object-oriented databases [14,15,13,22,25].[5]

---

[2] http://www.hassan-ait-kaci.net/hlt/doc/hlt/jaccdoc/000_START_HERE.html

[3] See Section 2.1.

[4] Other formally derived abstract machines like the Categorical Abstract Machine (CAM) also led to variants of formal compilation of functional languages (*e.g.*, Caml). This approach was also adopted for the chemical metaphor formalizing concurrent computation as chemical reaction originally proposed by Banâtre and Le Métayer [8] and later adapted by Berry and Boudol to define their Chemical Abstract Machine (ChAM) [9]. The same also happened for Logic Programming [3].

[5] That is, at the moment, our prototype Algebraic Query Language (AQL v0.00) is a functional language augmented with a calculus of compehensions *à la* Fegaras-Maier [22], or *à la* Grust [25]. In other words, it is a complete query language, powerful enough to express most of ODMG's OQL, and thus many of its derivatives such as, *e.g.*, XQuery [12] and

This machine was implemented and used by the author to generate several experimental 100%-java implementation of various language prototypes. Thus, what was actually implemented in this toolkit was done following a "by need" priority order. It is not so complete as to already encompass all the necessary building blocks needed for all known styles of programming and type semantics. It is meant as an open set of tools to be extended as the needs arise. For example, there is no support yet for $\mathcal{LP}$ [3], nor—more generally—$\mathcal{CLP}$ [27].

However, as limited as it may be, it already encompasses most of the basic familiar constructs from imperative and functional programming, including declarative aggregation (so-called "comprehensions"). Therefore, it is clearly impossible—not to say boring!—to cover all the nitty-gritty details of all the facets of the complete abtract machine generation system. This article is therefore organized as an informal stroll over the most interesting novel features or particularities of our design as it stands to date.

1.2 Our approach—abstract programming language design

The approach we follow is that of compiling a specific relatively more sophisticated outer syntax into a simpler instruction-based "*machine*" language. However, for portability, this inner language is that of an "*abstract*" machine. In other words, it is just an intermediate language that can be either interpreted more efficiently on an emulator of that abstract machine, and/or be mapped to actual instruction-based assembly code of a specific machine more easily.

Thus, as for most compiled typed programming languages, there are actually several languages:

- a *surface language*—the syntax used by users to compose programs;
- a *kernel language*—the "essential" language into which the surface language is normalized;
- a *type language*—the language describing the types of expressions;
- an *intermediate language*—the language that is executable on an instruction-based abstract machine.

Although we will not develop it into much detail in this paper, the Java execution backend for carrying out the operational semantics of the above *à la carte* design consists of:

- *An operational semantic language*—interpreting an abstract instruction set having effects on a set of runtime structures. The latter define the state of an execution automaton. The objects operated on and stored in these structures are the basic data representation all surface language constructs.

XPath [33], *etc.*, ... This version of AQL can be run both interactively and in batch mode. In the former case, a user can define top-level constructs and evaluate expressions. AQL v0.00 supports 2nd-order (ML-like) type polymorphism, automatic currying, associative arrays, multiple type overloading, dynamic operator overloading, as well as (polymorphic) type definition (both aliasing and hiding), classes and objects, and (of course) monoid homomorphisms and comprehensions (*N.B.*: no subtyping nor inheritance yet—but this is next on the agenda [23,24,10,11]).

– *A type-directed display manager*—maintaining a trace emulation of abstract machine code execution in relation to the source code it was generated from. This is also useful for debugging purposes while managing three sorted stacks (depending on the nature of Java data pushed on the various sorted stacks—`int`, `double`, or `Object`).[6]
– *A type-directed data reader*—management for reading three sorts of data (`int`, `double`, or `Object`).

The same applies for pragmatics as well:

– *Concrete vs. abstract error handling*—delegation of error reporting by inheritance along ⟨*design*⟩`.backend.Error.java` class hierarchy.[7]
– *Concrete vs. abstract vocabulary*—handling of errors according to the most specifically phrased error-handling messaging.

1.3 Organization of paper

The rest of this document is organized as follows. Section 2 overviews original generic syntax-processing tools that have been conceived, implemented, and used to ease the experimental front-end development for language processing systems. Section 3 gives a high-level description of the architectural attributes of a set of kernel classes of programming language constructs and how they are processed for typing, compiling, and executing. Section 4 discusses the type system, which is made operational as a polymorphic type inference abstract machine enabling multiple-type overloading, type encapsulation, object-orientation, and type (un)boxing analysis. Section 5 sums up the essentials of how declarative iteration over collections may be specified using the notion of monoid homomorphism and comprehension as used in object-oriented databases query languages to generate efficient collection-processing code. Section 6 concludes with a quick recapitulation of the contents and future perspectives.

In order to make this paper as self-contained as possible, the above overview of salient aspects of the system that has been implemented is followed by an Appendix of brief tutorials on essential key concepts and terminology this work relies upon, and/or extends.

---

[6] This is essentially a three-way SECD/FAM used to avoid systematically having to "box" into objects primitive Java values (*viz.*, of type `int` and `double`). This enables precious optimization that is particularly needed when dealing with variables of static polymorphic types but dynamically instantiated into `int` and `double` [32].

[7] Here and in what follows, we shall the following abbreviated class path notation:

- "⟨*syntax*⟩`.`" for "`hlt.language.syntax.`"
- "⟨*design*⟩`.`" for "`hlt.language.design.`" and this latter package's sub-packages:
  - "⟨*kernel*⟩`.`" for "⟨*design*⟩`.kernel.`"
  - "⟨*types*⟩`.`" for "⟨*design*⟩`.types.`"
  - "⟨*instructions*⟩`.`" for "⟨*design*⟩`.instructions.`"
  - "⟨*backend*⟩`.`" for "⟨*design*⟩`.backend.`"

when referring to actual classes' package paths.

2 Syntax Processing

2.1 Jacc—Just another compiler compiler

At first sight, Jacc may be seen as a "100% Pure Java" implementation of an LALR(1) parser generator [2] in the fashion of the well-known UNIX tool `yacc`—"yet another compiler compiler" [28]. However, Jacc is much more than... *just another compiler compiler*: it extends `yacc` to enable the generation of flexible and efficient Java-based parsers and provides enhanced functionality not so commonly available in other similar systems.

The fact that Jacc uses `yacc`'s metasyntax makes it readily usable on most `yacc` grammars. Other Java-based parser generators all depart from `yacc`'s format, requiring nontrivial metasyntactic preprocessing to be used on existing `yacc` grammars—which abound in the world, `yacc` being by far the most popular tool for parser generation. Importantly, Jacc is programmed in pure Java—this makes it fully portable to all existing platforms, and immediately exploitable for web-based software applications.

Jacc further stands out among other known parser generators, whether Java-based or not, thanks to several additional features. The most notable are:

- Jacc uses the most efficient algorithm known to date for its most critical computation (*viz.*, the propagation of LALR(1) lookahead sets). Traditional `yacc` implementations use the method originally developed by DeRemer and Penello [19]. Jacc uses an improved method due to Park, Choe, and Chang [34], which drastically ameliorates the method of by DeRemer and Penello. To this author's best knowledge, no other Java-based metacompiler system implements the Park, Choe, and Chang method [18].
- Jacc allows the user to define a complete class hierarchy of parse node classes (the objects pushed on the parse stack and that make up the parse tree: nonterminal and terminal symbols), along with any Java attributes to be used in semantic actions annotating grammar rules. All these attributes are accessible directly on any pseudo-variable associated with a grammar rule constituents (*i.e.*, `$$`, `$1`, `$2`, *etc.*).
- Jacc makes use of all the well-known conveniences defining precedences and associativity associated to some terminal symbols for resolving parser conflicts that may arise. While such conflicts may in theory be eliminated for any LALR(1) grammar, such a grammar is rarely completely obtainable. In that case, `yacc` technology falls short of providing a safe parser for non-LALR grammar. Yet, Jacc can accommodate any such eventual unresolved conflict using non-deterministic parse actions that may be tried and undone.
- Further still, Jacc can also tolerate non-deterministic tokens. In other words, the same token may be categorized as several distinct lexical units to be tried in turn. This allows, for example, parsing languages that use no reserved keywords (or more precisely, whose keywords may also be tokenized as identifiers, for instance).
- Better yet, Jacc allows dynamically (re-)definable operators in the style of the Prolog language (*i.e.*, at parse-time and run-time). This offers great flexibility for on-the-fly syntax customization, as well as a much greater recognition power, even

where operator symbols may be overloaded (*i.e.*, specified to have several precedences and/or associativity for different arities).

- Jacc supports partial parsing. In other words, in a grammar, one may indicate any nonterminal as a parse root. Then, constructs from the corresponding sublanguage may be parsed independently from a reader stream or a string.
- Jacc automatically generates a full HTML documentation of a grammar as a set of interlinked files from annotated `/**...*/` `javadoc`-style comments in the grammar file, including a navigatable pure grammar in "`yacc` form," obtained after removing all semantic and serialization annotations, leaving only the bare syntactic rules.
- Jacc may be directed to build a parse-tree automatically (for the concrete syntax, but also for a more implicit form which rids a concrete syntax tree of most of its useless information). By contrast, regular `yacc` necessitates that a programmer add explicit semantic actions for this purpose.
- Jacc supports a simple annotational scheme for automatic XML serialization of complex Abstract Syntax Trees (AST's) [6]. Grammar rules and non-punctuation terminal symbols (*i.e.*, any meaning-carrying tokens such as, *e.g.*, identifiers, numbers, *etc.*) may be annotated with simple XML templates expressing their XML forms. Jacc may then use these templates to transform the Concrete Parse Tree (CST) into an AST of radically different structure, constructed as a `jdom` XML document.[8] This yields a convenient declarative specification of a tree transduction process guided by just a few simple annotations, where Jacc's "sensible" behavior on unannotated rules and terminals works "as expected." This greatly eases the task of retargeting the serialization of a language depending on variable or evolving XML vocabularies.

With Jacc, a grammar can be specified using the usual familiar `yacc` syntax with semantic actions specified as Java code. The format of the grammar file is essentially the same as that required by `yacc`, with some minor differences, and a few additional powerful features. Not using the additional features makes it essentially similar to the `yacc` format.

For the intrigued reader curious to know how one may combine dynamic operator with a static parser generator, Section 2.2 explains in some detail how Jacc extends `yacc` to support Prolog-style dynamic operators.

2.2 LR-parsing with dynamic operators

In this section, we explain, justify, and specify the modifications that need to be made to a classical table-driven LALR(1) parser generator *à la* `yacc` [28]. For such a compiler generator to allow Prolog-style dynamic operators, it is necessary that it be adapted to account *statically* (*i.e.*, at compile-time) for runtime information. Indeed, in Prolog, operators may be declared either at compile-time or at runtime using the built-in predicate `op/3`.[9]

[8] `http://www.jdom.org/`

[9] See Appendix Section A for a quick review of Prolog-style dynamic operators.

How Jacc enables static LR-parsing with dynamic operators In an LR-parser such as one generated by yacc, precedence and associativity information is no longer available at parse-time. It is used statically at parser generation-time to resolve potential conflicts in the parser's actions. Then, a fixed table of unambiguous actions is passed to drive the parser, which therefore always knows what to do in a given state for a given input token.

Thus, although they can recognize a much larger class of context-free languages, conventional shift-reduce parsers for LR grammars cannot accommodate parse-time ambiguity resolution. Although this makes parsing more efficient, it also forbids a parser generated by a yacc-like parser generator to support Prolog style operators.

In what follows, we propose to reorganize the structure of the implementation of a yacc-style parser generator to accommodate Prolog-style dynamic operators. We do so:

- increasing the user's convenience to define and use new syntax dynamically without changing the parser;
- adding new features while preserving the original yacc metasyntax;
- retaining the same efficiency as yacc-parsing for grammars which do not use dynamic operators;
- augmenting the recognizing power of bottom-up LALR parsing to languages that support dynamically (re)definable operators;
- making full use of the object-oriented capabilities of Java to allow the grammar specifier to tune the parser generation using user-defined classes and attributes.

Declaring dynamic operators The first issue pertains to the way we may specify how dynamic operators are connected with the grammar's production rules. The command:

```
%dynamic op
```

is used to declare that the parser of the grammar being specified will allow defining, or redefining, dynamic operators of category op. The effect of this declaration is to create a non-terminal symbol named op that stands for this token category. Three implicit grammar rules are also defined:

```
op : 'op_' | '_op_' | '_op' ;
```

which introduce, respectively, prefix, infix, and postfix, subcategories for operators of category op. These are terminal symbols standing as generic tokens that denote specific operators for each fixity. Specific operators on category op may be defined in the grammar specification as follows:

```
%op <operator> <specifier> <precedence>
```

For example,

```
%op '+' yfx 500
```

declares the symbol '+' to be an infix binary left-associative operator of category op, with binding tightness 500, just as in Prolog.

In addition, the generated parser defines the following method:

```
public final static void op ( String operator
                            , String specifier
                            , int precedence)
```

whose effect is to define, or redefine, an operator for the token category op dynamically using the given (Prolog-style) specifier and (Prolog-style) precedence. It is this method that can be invoked in a parser's semantic action at parse time, or by the runtime environment as a static method.

An operator's category name may be used in a grammar specification wherever an operator of that category is expected. Namely, it may be used in grammar rules such as:

```
expression : op expression
           | expression op
           | expression op expression
           ;
```

Using the non-terminal symbol op in a rule such as above allows operators of any fixity declared in the op category to appear where op appears. However, if an occurrence must be limited to an op of specific fixity only, then one may use:

- 'op_' for a prefix operator of category op;
- '_op' for a postfix operator of category op;
- '_op_' for an infix operator of category op.

For example, the above rules can be better restricted to:

```
expression : 'op_' expression
           | expression '_op'
           | expression '_op_' expression
           ;
```

A consequence of the above observations is that a major modification in the parser generator and the generic parser must also be made regarding the parser actions they generate for dynamic operators. A state may have contending actions on a given input. Such a state is deemed conflictual if and only if the input creating the conflict is a dynamic operator, or if one of its conflicting actions is a reduction with a rule whose tag is a dynamic operator. All other states can be treated as usual, resolving potential conflicts using the conventional method based on precedence and associativity. Clearly, a dynamic operator token category does not have this information but delegates it to the specific token, which will be known only at parse time. At parser-construction time, a pseudo-action is generated for conflictual states which delays decision until parse time. It uses the state's table associating a set of actions with the token creating the conflict in this state. These sets of conflicting actions are thus recorded for each conflictual state.

When a token is identified and the current state is a conflictual state, which action to perform is determined by choosing in the action set associated to the state according to the same disambiguation rules followed by the static table construction but using the current precedence and associativity values of the specific operator being read. If a "`reduce`" action in the set involves a rule tagged with a dynamic operator, which precedence and associativity values to use for the rule are those of the specific operator tag for that rule, which can be obtained in the current stack. The stack offset of that operator will depend on which of the dynamic operator's rules is being considered.

Ambiguous tokens Note that in general, the tokenizer may return a set of possible tokens for a single operator. Consider for example the following grammar:

```
%token '!'
%dynamic op1
%op1 '!' yf 200
%dynamic op2
%op2 '!' yfx 500
%%
expression : '!' expression
           | expression _op1
           | expression _op2 _ expression
           ;
%%
```

For this grammar, the character '`!`' may be tokenized as either '`!`', '`op1`', or '`op2`'. The tokenizer can therefore be made to dispense with guaranteeing a token's lexical category. Looking up its token category tables, the parser then determines the set of admissible lexical categories for this token in the current state (*i.e.*, those for which it has an action defined). If more than one token remain in the set, a choice point for this state is created. Such a choice point records the current state of parsing for backtracking purposes. Namely, the grammar state, and the token set. The tokens are then tried in the order of the set, and upon error, backtracking resets the parser at the latest choice point deprived of the token that was chosen for it.

Note that the use of backtracking for token identification is not a guarantee of complete recovery. First, full backtracking is generally not a feasible nor desirable option as it would entail possibly keeping an entire input stream in memory as the buffer grows. The option is to keep only a *fixed-size buffer* and flush from the choice point stack any choice point that becomes stale when this buffer overflows. In effect, this enforces an automatic commit whenever a token choice is not invalidated within the time it takes to read further tokens as allowed by the buffer size.

Second, although backtracking restores the parser's state, it does not automatically undo the side effects that may have been performed by the execution of any semantic action encountered between the failure state and the restored state. If there are any, these must be undone manually. Thus, Jacc allows specifying undo actions to be executed when a rule is backtracked over.

The only limitation—shallow backtracking—is not serious, and in fact the choice-point stack's size can be specified arbitrarily large if need be. Moreover, any input that

overuns the choice-point stack's default depth is in fact cleaning up space by getting rid of older and less-likely-to-be-used choice-points. Indeed, failure occurs generally shortly after a wrong choice has been made. We give separately a more detailed specification of the implementation of the shallow backtracking scheme that is adequate for this purpose.

Token declarations In order to declare tokens' attributes in `yacc`, one may use the commands `%token`, `%right`, `%left`, and `%nonassoc`. These commands also give the tokens they define a precedence level according to the order of declarations, tokens of equal precedence being declared in the same command. Since we wish to preserve compatibility with `yacc`'s notations and conventions, we keep these commands to have the same effect. Therefore, these commands are used as usual to declare static tokens. However, we must explicate how the implicit precedence level of static token declarations may coexist with the explicit precedence information specified by the Prolog-like dynamic operator declarations.

We also wish to preserve compatibility with Prolog's conventions. Recall that the number argument in a Prolog '`op/3`' declaration denotes the binding tightness of the operator, which is inversely related to parsing precedence. The range of these numbers is the interval $[1, 1200]$. To make this compatible with the foregoing `yacc` commands, the $\langle syntax \rangle$`.Grammar.java` class defines two constants:

```
static final int MIN_PRECEDENCE = 1;
static final int MAX_PRECEDENCE = 1200;
```

In order to have the binding tightness to be such that 1200 corresponds to minimum precedence and 1 to maximum precedence, we simply define the precedence level of binding tightness n to be $1200 - n + 1$. Thus, a declaration such as:

```
%op '+' yfx 500
```

assigns to binary '+' a precedence level of 701 (*viz.*, $1200 - 500 + 1$).

We also allow dynamic operators to be declared with the form:

```
%op <operator> <specifier>
```

leaving the precedence implicit, and defaulting to the precedence level effective at the command's execution time.

The first encountered token declaration with implicit precedence (*i.e.*, a conventional `yacc` token command or a two-argument dynamic operator command) uses the initial precedence level set to a default,[10] then increments it by a fixed increment. This increment is 10 by default, but the command:

```
%precstep <number>
```

may be used to set the increment to the given number. This command may be used several times. Each subsequent declaration with implicit precedence uses the current precedence level, then increments the precedence level by the current precedence increment. Any attempt to set a precedence level outside the $[1, 1200]$ range is ignored: the closest bound is used instead (*i.e.*, 1 if less and 1200 if more), and a warning is issued.

[10] This value is a system constant called $\langle syntax \rangle$`.Grammar.MIN_PRECEDENCE`.

3 The Kernel Language

A language construct is said to be *primitive* (or "*built-in*") if it is not expressed in terms of other language constructs.[11] The kernel language is the set of primitive language constructs. It is sometimes also called the "desugared" language. This is because non-primitive constructs that are often-used combinations of primitive stuctures are both easier to use and read by human programmers. Hence, before being given any meaning, a program expressed using the "sugared" language syntax is first translated into its equivalent "desugared" form in the kernel language containing only primitive expressions.

3.1 Processing a kernel expression

Fig. 1 gives the complete processing diagram from reading a ⟨*kernel*⟩`.Expression` denoting a program to executing it.

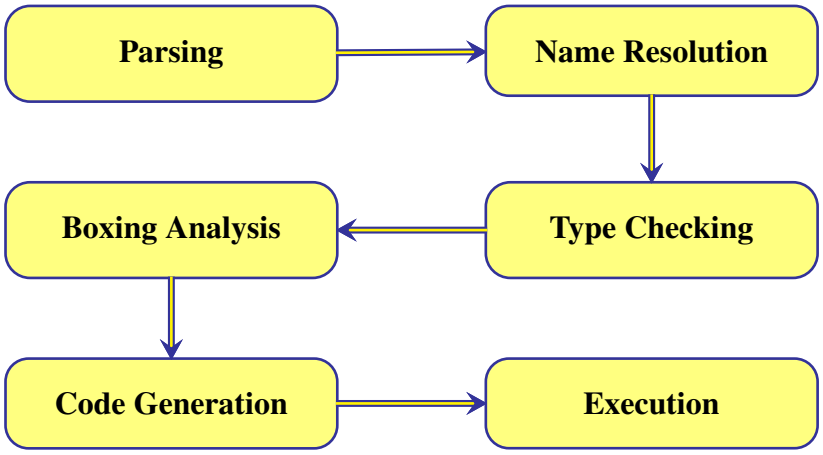

Fig. 1. Processing diagram

Typically, upon being read, such a ⟨*kernel*⟩`.Expression` will be:

1. "*name-sanitized*"—in the context of a ⟨*kernel*⟩`.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 ⟨*types*⟩`.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);

[11] This does not mean that it could not be. It just means that it is provided natively, either to ease oft-used syntax, and/or make it more efficient operationally.

3. "*sort-sanitized*"—in the context of a ⟨*kernel*⟩`.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 of the three type sorts (`int`, `double`, `Object`);
4. *compiled*—in the context of a ⟨*kernel*⟩`.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 ⟨*backend*⟩`.Runtime` denoting the appropriate runtime environment in the context of which to execute its sequence of instructions.

The syntax sanitizer A *sanitizer* is an object that "cleans up"—so to speak—an expression of any possibly remaining ambiguities as it is being parsed and further 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.[12]

Thus, a sanitizer is a discriminator of names and sorts.[13]

The type checker 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 ⟨*kernel*⟩`.Expression.TypeCheck(`⟨*types*⟩`.TypeChecker)` method that encodes its formal typing rule.

The compiler This is the class defining a compiler object. Such an object serves as the common compilation context shared by an ⟨*kernel*⟩`.Expression` and the subexpressions comprising it. Each type of expression representing a syntactic construct of the kernel language defines a ⟨*kernel*⟩`.Expression.compile(`⟨*kernel*⟩`.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 a sequence of instructions, each of specific subtype of abstract type ⟨*instructions*⟩`.Instruction`, corresponding to a top-level expression and its subexpressions.

---

[12] These offsets are the so-called *de Bruijn* indices of λ-calculus [30]—Or rather, their sorted version.

[13] It has occurred to this author that the word "*sanitizer*" is perhaps a tad of a misnomer. Perhaps "*discriminator*" might have been a better choice. This also goes for the ⟨*kernel*⟩`.Sanitizer.java` class' method names (*i.e.*, `discriminateNames` and `discriminateSorts` rather than `sanitizeNames` and `sanitizeSorts`).

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 ⟨*backend*⟩`.Runtime` object, or, in the case of a ⟨*kernel*⟩`.Definition`, be saved in the code array in the ⟨*kernel*⟩`.Definition`'s ⟨*kernel*⟩`.codeEntry()` field of type ⟨*types*⟩`.DefinedEntry`, which is an object that encapsulates its code entry point, and which may in turn then be used to access the defined symbol's code for execution.

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

In Appendix Section B, this process is illustrated in more detail on a few typical as well as less typical expressions.

4 Types

> We have illustrated a style of programming based on the use of rich type systems. This is not new in general, but the particularly rich type system we have described, based on type quantifiers and subtypes, extends the state of the art. This rich type structure can account for functional, imperative, algebraic, and object-oriented programming in a unified framework, and extends to programming in the large and, with care, to system programming.
>
> LUCA CARDELLI—"Typeful Programming" [17]

4.1 Type language

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

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. The syntax of types is defined 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.[14]

[14] Or more precisely that $\forall$ never occurs nested inside a function type (*i.e.*, in a _ $\rightarrow$ _ type). This apparently innocuous detail ensures decidability of type inference. Incidentally, the 2*nd* order

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.

To this author's knowledge, there is no established prevailing technology for supporting *both* ML-polymorphic type inference and multiple type overloading. So here, as in a few other parts of this overall design, I have had to innovate. I essentially implemented a type proving logic using techniques from (Constraint) Logic Programming in order to handle the combination of types supportable by this architecture.

Currying Currying is an operation that exploits the following mathematical isomorphism of types:[15]

$$T, T' \to T'' \;\simeq\; T \to (T' \to T'') \tag{1}$$

which can be generalized for a function type of any number of arguments to any of its multiple curryed forms—*i.e.*, for all $k \;=\; 1, \ldots, n-1$:

$$T_1, \ldots, T_n \to T \;\simeq\; T_1, \ldots, T_k \to (T_{k+1}, \ldots, T_n \to T) \tag{2}$$

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.

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 (1) as a rewrite rule, although *backwards* (*i.e.*, from right to left) as long as it applies. The second phase (renaming) consists in making a consistent copy of all types reachable from a type's root.

---

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.

[15] 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 (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.

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.

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.

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.

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*?[16] This problem was addressed by Xavier Leroy and he proposed a solution, which has been implemented in the CAML compiler [32].[17] 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.

For our purpose, the main idea from Leroy's solution was adapted and improved 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.

[16] The alternative would be either to compile distinct copies for all possible runtime sort instantiations (like, *e.g.*, C++ template functions), or compiling each specific instantiation as it is needed. The former is not acceptable because it tends to inflate the code space explosively. The latter can neither be envisaged because it goes against a few (rightfully) sacrosanct principles like separate compilation and abstract library interfacing—imagine having to recompile a library everytime you want to use it!

[17] See `http://caml.inria.fr/`.

4.2 Type processing

The type system consists of two complementary parts: a *static* and a *dynamic* part.[18] 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 appropriate 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 integers);
  - `Real` range extension type (close interval of floating-point numbers).

  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+`, `double+`).

Static types The static type system is the part of the type system that is effective at compile-time.

Primitive types

- `Boxable` types (Void, Int, Real, Char, and Boolean)
- `Boxed` types (*i.e.*, boxed versions of `Boxable` types or non-primitive types)

Non-primitive types

- Built-in type constants (*e.g.*, String, *etc.*, . . .)
- Type constructors
- Function types
- Tuple types:
  - Position tuple types

[18] For the complete class hierarchy of types in the package ⟨*design*⟩`.types`, see Fig. 2.

    - Named tuple types
- Array types:
    - 0-based int-indexed arrays
    - `Int` range-indexed arrays
    - `Set`-indexed arrays
    - Multidimensional arrays
- Collection types (`Set`(α), `Bag`(α), and `List`(α)).
- `Class` types

The `Class` type 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:

$$\mathfrak{class}\ \texttt{classname}\ \{\ \textit{interface}\ \}\ [\ \{\ \textit{implementation}\}\ ] \tag{3}$$

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:

$$\begin{array}{l}\mathfrak{class}\ \texttt{Counter}\ \{\ \texttt{value} : \mathfrak{Int} = 1;\\ \qquad\qquad\quad \mathfrak{method}\ \texttt{set} : \mathfrak{Int} \to \texttt{Counter};\\ \qquad\qquad \}\\ \qquad\qquad \{\ \texttt{set}(\texttt{value} : \mathfrak{Int}) : \texttt{Counter}\\ \qquad\qquad\quad = (\mathfrak{this}.\texttt{value} = \texttt{value});\\ \qquad\qquad \}\end{array} \tag{4}$$

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 $\mathfrak{Void}$,[19] 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

[19] 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.

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:

$$\mathfrak{Def}\ \texttt{set}(\texttt{x} : \texttt{Counter}, \texttt{n} : \mathfrak{Int}) : \texttt{Counter} \ = \ (\texttt{x.value} = \texttt{n}); \tag{5}$$

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:

$$\texttt{c} = \mathfrak{new}\ \texttt{Counter}; \tag{6}$$

the slot that corresponds to the location of the `value` field will be initialized to the value 1 of type $\mathfrak{Int}$. Then, field and method invocation can be done using the familiar "dot notation;" *viz.*:

$$\begin{array}{l}\texttt{c.set}(\texttt{c.value} + 2);\\ \texttt{write}(\texttt{c.value});\end{array} \tag{7}$$

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

$$\begin{array}{l}\texttt{set}(\texttt{c}, \texttt{value}(\texttt{c}) + 2);\\ \texttt{write}(\texttt{value}(\texttt{c}));\end{array} \tag{8}$$

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) could be specified more simply as:

$$\{\ \texttt{set}(\texttt{n}) = (\texttt{value} = \texttt{n});\ \} \tag{9}$$

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 ⟨*types*⟩`.Tables.java`; this ensures that its type definition links it to an appropriate `ClassType` object; namely, a class structure represented by an object of type ⟨*types*⟩`.ClassInfo.java` where the 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 an object of type `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).

The type system Fig. 2 shows the hierarchy of Java classes representing the categories of types currently comprising the type system. The classes represented in boxes are abstract classes. There could be more, of course.

Structure of `TypeChecker` An object of the class ⟨*types*⟩`.TypeChecker.java` 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 ⟨*types*⟩`.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.[20] 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 ⟨*kernel*⟩`.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.

[20] See Appendix Section B.

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

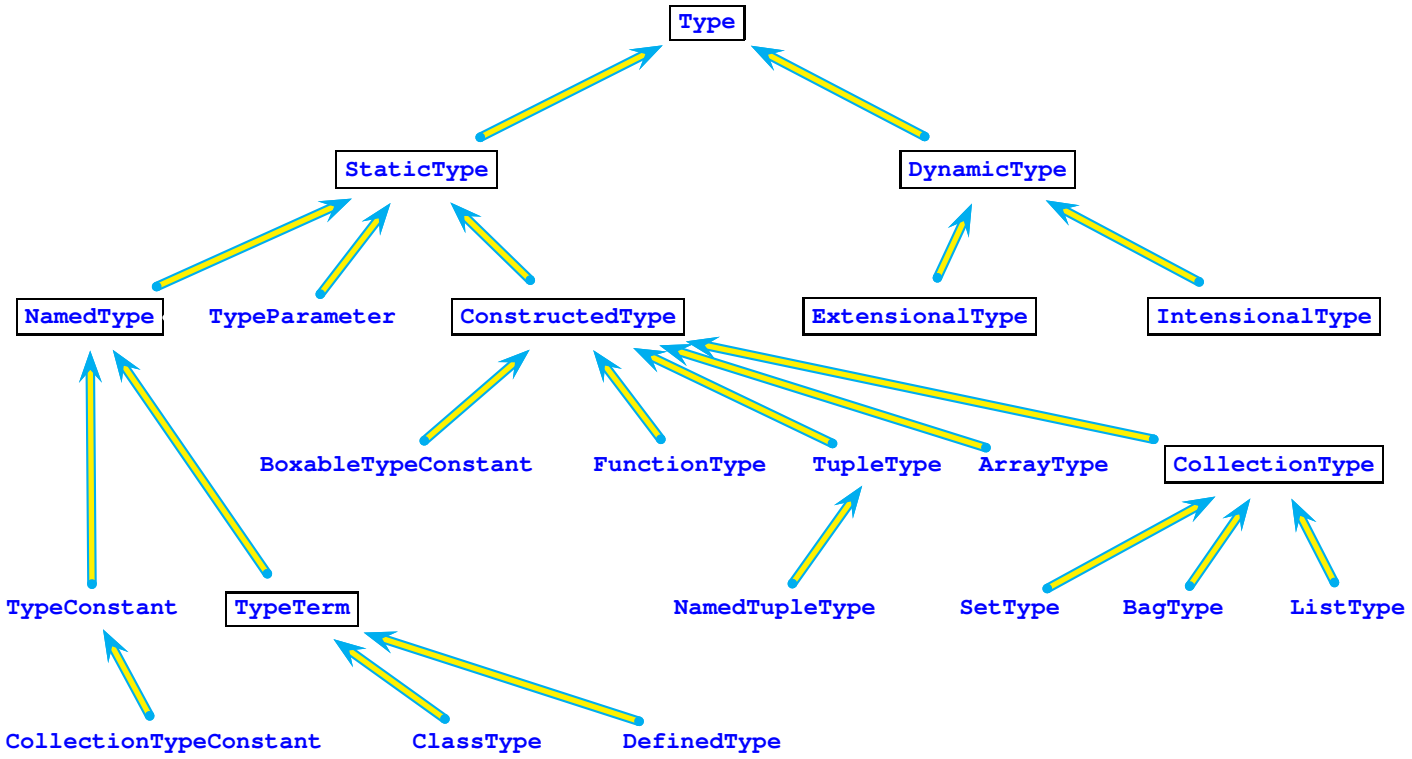

Fig. 2. The type system—Metaclass Hierarchy

This amounts to the on-the-fly compiling of type-checking rules into "typing-goal" instructions that must be stored for potential retrial upon subsequent failure. Fig. 3 lists some typing goals making up the instruction set of the type inference abstract machine generated by the type checker.

- `EmptyGoal`
- `TypingGoal`
- `UnifyGoal`
- `GlobalTypingGoal`
- `SubTypeGoal`
- `BaseTypeGoal`
- `ArrayIndexTypeGoal`
- `PruningGoal`
- `PushExitableGoal`
- `PopExitableGoal`
- `CheckExitableGoal`
- `ResiduatedGoal`
- `ShadowUnifyGoal`
- `UnifyBaseTypeGoal`
- `NoVoidTypeGoal`

Fig. 3. Typing goals instruction set for the type inference abstract machine

In order to coordinate type proving in a common context, the same 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 $\langle types \rangle$`.TypeChecker` object are:

- a *goal stack* containing *goal* objects (*e.g.*, $\langle types \rangle$`.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 $\langle types \rangle$`.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 from where to constructs new `TypingGoal` objects to try upon failure;
  - pointers to all trails up to which to undo effects.

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 (1) convenience of making programs more legible by giving terser "logical" names (or terms) to otherwise verbose type expressions, and (2) that of hiding information details of a

type and 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 ⟨*types*⟩`.Tables.java` 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 ⟨*types*⟩`.Tables.java.typeTable` variable contains the naming table for types and the ⟨*types*⟩`.Tables.java.symbolTable` variable contains the naming table for other (non-type) global names.

This section will overview some type-related data-structures starting from the class that manages symbols: ⟨*types*⟩`.Tables.java`. The names can be those of types and values. They are *global* names.[21] The type namespace is independent of the value namespace—*i.e.*, the same name can denote a value and a type.

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, as soon as a type's structure depends on a runtime value, it is necessarily a dynamic type. These are also often referred to as *dependent* types. For example, `array_of_size(int n)`, where `n` is the size of the array and is a runtime value. It denotes a "safe" array type depending on the array size that may be only computed at runtime.[22] From this, we require that a class implementing the `DynamicType` interface provides a method:

```
public boolean verifyCondition ()
```

that is invoked systematically by code generated 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:

[21] At the moment, there is no name qualification or namespace management. When this service is provided, it will also be through the ⟨*types*⟩`.Tables.java` class.

[22] *e.g.*, *à la* Java arrays.

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

We shall consider here a few such dynamic types (motivated esssentially by the typing needs of for OPL, or similar constraint languages [26]). 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 integer range, a floating-point number 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. This method simply 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 constitute "contracts" that must be honored as soon as the type's extension becomes known (either positively—removing the honored contract; or, negatively—causing a type error).

Extensional types that have been included are set types, range types (integer and floating-point), and enumeration types. Other dynamic types could of course be added as needed (*e.g.*, lists, bags, *etc.*).

Intensional types can be accommodated by defining new opaque types—*e.g.*, in order to define non-negative numbers, we introduce a new (opaque) type `Nat` as a dynamically constrained `Int` type whose `verifyCondition` method ensures that only non-negative integer values may be used for this type.

5 Computing with Collections

There are two classes defined for such expressions: ⟨*kernel*⟩`.Homomorphism.java` and ⟨*kernel*⟩`.Comprehension.java`. These classes are based on the formal notion of monoid homomorphisms and comprehension as defined in query-language formalisms [14,15,13,22].[23]

These two classes of expressions use monoid homomorphisms as declarative iterators. Thus, henceforth, by *homomorphism* we 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 given in Fig. 4. Monoid homomorphisms are quite useful for expressing a certain kind of iteration declaratively.

[23] See Appendix Section E for a refresher on monoid homomorphisms and comprehensions.

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 $[]$
...		

Fig. 4. Examples of some familiar monoids

The class `Homomorphism` is the class of objects denoting (monoid) homomorphisms. An instance of such a class defines all the needed parameters for representing and iterating 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:

$$\begin{aligned} \mathbf{hom}^{\mathbb{1}_\oplus}_{\oplus}(f)[] &= \mathbb{1}_\oplus \\ \mathbf{hom}^{\mathbb{1}_\oplus}_{\oplus}(f)[H|T] &= f(H) \oplus \mathbf{hom}^{\mathbb{1}_\oplus}_{\oplus}(f)T \end{aligned} \tag{10}$$

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 conceptually 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 $T' \to coll(T)$; and,
- the operation "adding" an element to a collection—of type $T, coll(T) \to coll(T)$.

A *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 $T' \to T$; and,
- the monoid operation—of type $T, T \to T$.

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 $\mathsf{fun}\ x\ \cdot\ x \oplus \mathbb{1}_{\oplus}$ (*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 suppression 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. This is what the kernel class `Comprehension` encapsulates, as it is defined relying on the class `Homomorpism`, exactly as its formal definition does.

A monoid comprehension is an expression of the form:

$$\langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid q_1, \ldots, q_n \} \tag{11}$$

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:

$$\begin{aligned}
\langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid \} &\stackrel{\text{def}}{=} e \oplus \mathbb{1}_{\oplus} \\
\langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid x \leftarrow e', Q \} &\stackrel{\text{def}}{=} \mathfrak{hom}_{\oplus}^{\mathbb{1}_{\oplus}} [\lambda x. \langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid Q \}] (e') \\
\langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid c, Q \} &\stackrel{\text{def}}{=} \mathfrak{if}\ c\ \mathfrak{then}\ \langle \oplus, \mathbb{1}_{\oplus} \rangle \{ e \mid Q \}\ \mathfrak{else}\ \mathbb{1}_{\oplus}
\end{aligned} \tag{12}$$

In other words, a comprehension is fully expressible in terms of compositions of homomorphims. 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 [39,21]).

Although a monoid comprehension can be effectively computed using nested loops (*i.e.*, using a simple iteration semantics), 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 [14,15,39,13,21]. All the attributes of the syntax of monoid comprehensions derived from monoid homomorphisms are represented in these type classes.

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.

6 Conclusion

6.1 Recapitulation

In this document we summarized the main characteristics of an abstract, reusable, and extensible programming language architecture, and its implementation in Java. We overviewed original generic syntax-processing tools that have been conceived, implemented, and used to ease the experimental front-end development for language processing systems. This consisted of Jacc, a flexible metacompiler all done in 100%-pure Java. We explained the machinery needed to extend LALR-parsing to enable dynamic operators *à la* Prolog. We gave a high-level description of the architectural attributes of a set

of kernel classes of programming language constructs and how they are processed for typing, compiling, and executing. We presented our architecture general processing diagram taking a kernel expression into straightline abstract-machine code. We discussed a type system that is the basis for a polymorphic type inference abstract machine enabling multiple-type overloading, type encapsulation, object-orientation, and type (un)boxing analysis. We described the type language primitives and constructors, and how they were analyzed for efficient code generation and execution. We explained our implementation of type-checking and how execution of declarative iteration over collections may be specified using the notion of monoid homomorphism and comprehension as used in object-oriented database query languages to generate efficient collection-processing code.

For the sake of making this document self-contained, we append below a set of sections of tutorial nature giving background material and finer-point discussions regarding what was presented.

6.2 What's next?

This architecture offers a compromise between formal executable specification systems (*e.g.*, [38,10]) and pragmatic needs for practical language prototyping backward compatible with popular existing tools (`yacc`, Java), while staying an *extensible* system—a *poor man's language kit?...* It enables fast and low-cost development of programming languages with basic and advanced features using familiar programming idioms like `yacc` and Java with a relatively high efficiency and confidence of correctness.

Importantly, it is *open* and favors *ease of extension* as well as *interoperability* with popular representation standards such as the W3C's. As mentioned several times, and made explicit in the title, this is work to be continued. Indeed, more tools and capabilities are to be added as this author's sees the need. The system has shown itself a practical and useful experimental tool. However, much more remains to be done (*e.g.*, namespace and access management, rule-based programming, logic programming, finer type logics, *etc.*, ...). Here are a few of the most immediate on our agenda.

- *Notation*—The next step is to extend Jacc by providing other structure-generating options besides XML, such as the JavaScript Object Notation (JSON)[24] and its version for Linked Data (JSON-LD).[25] With this tool, it will then be easier to experiment using Jacc to generate RDF-triples (or variations thereof) as compilation schemes from high-level (*i.e.*, more legible and user-friendly) KR languages (such as, *e.g.*, $\mathcal{OSF}$ or LIFE syntax—or even higher level; *e.g.*, NL dialects).
- *Typing*—Truly polymorphic object-oriented subtyping *à la* Gesberg, *et al.* [23,24], or Satisfiability Modulo Theories *à la* Bierman *et al.* [10,11]. This is indeed a most desired set of type-analytical capabilities to enable subtyping and class inheritance in our type logic. The type-checking rules given for these systems are the best candidates to use for this objective.

[24] http://www.json.org/

[25] http://json-ld.org/

– *Semantics*— The most ambitious next step in terms of semantics, would be to extend the current design with additional abstract meta-constructs for $\mathcal{LP}$ [3] and $\mathcal{CLP}$ [27] (and LIFE [4,7] in particular).
– *Pragmatics*—Not much has been said about the backend system.[26] Among the most desired to be done is a graphical front end based on Eclipse.[27] Wrapping all the backend tools and services in such a front-end would greatly help further meta-development.
– *Implementation*—Once abstracted into stable interfaces, any design may then be made more efficient where needed since implementation has thus been made independent. Attention may then be safely given to clever optimization of any type of algorithms used in the implementation of these interfaces, relying on time-tested techniques [1].

Appendix

In order to make this article self-contained, we include next a set of tutorials that overview essential background notions. Thus, this appendix consists of the following sections. Section A recalls the peculiar way that Prolog uses to enable changing the syntactic properties of its operators dynamically—*i.e.*, at run time. Section B describes how a few familiar programming language contructs may be specified as classes of objects and how these classes are processed in various syntax, typing, or execution contexts. Section C recounts notions on algebraic monoids. Section D is a reminder of the abstract syntax and type inference logic for a basic typed polymorphic λ-calculus with tupling. Section E presents OQL, an Object Query Language extending this basic λ-calculus into a monoid comprehension calculus dealing with collection data in a declarative manner thanks to monoid homomorphisms. Section F is a brief specification of the backend tooling needed to complete the system,

A Prolog-style Dynamic Operators

In Prolog, the built-in operator '`op/3`' offers the user the means to declare or modify the syntax of some of its operators. For example, as will be explained below:

```
?- op(500,yfx,+).
```

declares the symbol '`+`' to be an infix binary left-associative operator with binding tightness 500. The second argument of the built-in predicate `op/3` is called the operator's *specifier*. It is a symbol that encodes three kinds of information concerning the operator; namely:

– arity (*unary* or *binary*),
– "fixity" (*prefix*, *infix*, or *postfix*),
– associativity (*left-*, *right-*, or *non-associative*).

[26] See Appendix Section F
[27] http://www.eclipse.org/

The specifier is an identifier consisting of either two or three of the letters '`f`', '`x`', and '`y`', which are interpreted as follows. The letter '`f`' stands for the operator's position in an expression (its *fixity*), and the letters '`x`' and '`y`' stand for the arguments' positions. These letters are mnemonics for "*functor*," ('`f`') "*yes*," ('`y`') and "*no*" ('`x`'). A '`y`' occurring on the left (resp., right) of '`f`', means that the operator associates to the left (resp., right). An '`x`' occurring on the left (resp., right) of '`f`', means that the operator does not associate to the left (resp., right). Thus, the possible operator specifiers are shown in Table 1.[28]

Specifier	Arity	Fixity	Associativity
`fx`	unary	prefix	non-associative
`fy`	unary	prefix	right-associative
`xf`	unary	postfix	non-associative
`yf`	unary	postfix	left-associative
`xfx`	binary	infix	non-associative
`xfy`	binary	infix	right-associative
`yfx`	binary	infix	left-associative

Table 1. Mnemonic operator specifiers in Prolog

The binding tightness used by Prolog's '`op/3`' works in fact as the opposite of the precedence level used in parsing: the smaller a Prolog operator's binding tightness measure is, the more it takes precedence for parsing. These binding tightness measures range inclusively from 1 (maximum precedence) to 1200 (minimum precedence).

The third argument of '`op/3`' can be any syntactically well-formed Prolog functor. In particular, these need not be known as operator prior to runtime. Prolog's tokenizer only recognizes such a token as a functor. Thus, any functor, whether declared operator or not, can always be parsed as a prefix operator preceding a parenthesized comma-separated sequence of arguments. Whether it is a declared operator determines how it may be parsed otherwise. In Sicstus Prolog, for example:

```
| ?- X = 1 + 2.
X = 1+2 ?
yes
| ?- X = +(1,2).
X = 1+2 ?
yes
```

Prolog's parser can accommodate dynamic operators for two reasons:

[28] Note that '`yfy`' is not allowed as an operator specifier because that would mean an ambiguous way of parsing the operator by associating either to the left or to the right.

1. The syntax of Prolog is completely uniform - there is only one syntactic construct: the first-order term. Even what appear to be punctuation symbols are in fact functors (*e.g.*, '`:-`', '`,`', '`;`', *etc.*, ...). Indeed, in Prolog everything is either a logical variable or a structure of the form $f(t_1, \ldots, t_n)$.
2. Prolog parser's is an operator-precedence parser where precedence and associativity information is kept as a dynamic structure.[29]

Operator-precedence parsing is a bottom-up shift-reduce method that works simply by shifting over the input looking for a handle in a sentential form being built on the stack, and reducing when such a handle is recognized. A handle is the substring of a sentential form whose right end is the leftmost operator whose following operator has smaller precedence, and whose left end is the rightmost operator to the left of this right-end operator (inclusive), whose preceding operator has smaller precedence. This substring includes any nonterminals on either ends. For example, if '`*`' has higher precedence than '`+`', the handle in '`E + E * E + E`' is '`E * E`'.

Operator-precedence parsing is possible only for a very restricted class of grammars - the so-called "*Operator Grammars*." A context-free grammar is an Operator Grammar if and only if no production's right-hand side is empty or contains two adjacent nonterminals. For example, the grammar:

```
E : 'id' | P E | E O E | '(' E ')' ;
P : '-' ;
O : '+' | '*' | '-' | '/' ;
```

is not an operator grammar. But the equivalent grammar:

```
E : 'id' | '-' E | E '+' E | E '*' E | E '-' E
  | E '/' E | '(' E ')' ;
```

is. It is not difficult to see that a Prolog term can easily be recognized by an operator grammar. Namely,

```
T : 'var' | 'fun' | 'fun' '(' B ')'
  | 'fun' T | T 'fun' | T 'fun' T | '(' T ')' ;
B : T | T ',' B ;
```

which can thus easily accommodate dynamic operators.

B Structure of Kernel Expressions

The class ⟨*kernel*⟩`.Expression.java` 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);
- `Abstraction`: functional abstraction (*à la* λ-calculus);

[29] See "*the Dragon Book*," [2]—Section 4.6, pp. 203–215.

- `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;
- `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 traditional object-oriented "dot" dereferencing 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;

To illustrate the process, we next describe a few kernel constructs. A kernel expression description usually consist of some of the following items:

- ABSTRACT SYNTAX—describes the abstract syntax form of the kernel expression.

- OPERATIONAL SEMANTICS—for unfamiliar expressions, 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 more formally how a type should be verified or inferred using formal rules *à la* Plotkin's Structural Operational Semantics for typing the kernel expression, whose notation is briefly recalled as follows [35,36].
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:

$$\Gamma[x : T](y) \stackrel{\text{def}}{=} \begin{cases} T & \text{if } y = x; \\ \Gamma(y) & \text{otherwise.} \end{cases} \tag{13}$$

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

$$\frac{J_1, \ldots, J_n}{J} \tag{14}$$

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:

$$\frac{J_1, \ldots, J_n}{J} \ \texttt{if}\ c(J_1, \ldots, J_n) \tag{15}$$

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:

$$\frac{\Gamma_1 \vdash e_1 : T_1,\ \ldots,\ \Gamma_n \vdash e_n : T_n}{\Gamma \vdash e : T} \tag{16}$$

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 (15) looks like:

$$\frac{\Gamma_1 \vdash e_1 : T_1,\ \ldots,\ \Gamma_n \vdash e_n : T_n}{\Gamma \vdash e : T}\ \textbf{\textit{if}}\ \ cond(\ \Gamma,\ \Gamma_1, \ldots, \Gamma_n,\ e,\ e_1,\ \ldots, e_n,\ T, T_1, \ldots, T_n) \tag{17}$$

where "$cond(\Gamma, \Gamma_1,\ \ldots,\ \Gamma_n, e, e_1,\ \ldots,\ e_n, T, T_1,\ \ldots,\ T_n)$" 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*** $\ldots$*, 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:

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

Similarly, a *typing axiom*:

$$\frac{}{\Gamma \vdash e : T} \tag{18}$$

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:

$$\frac{}{\Gamma \vdash e : T}\ \textbf{\textit{if}}\ \ c(\Gamma, e, T) \tag{19}$$

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*."*

– COMPILING RULE—describes the way the expression's components are mapped into a straightline sequence of instructions. The compiling rule for expression e is given as a function $\mathfrak{compile}[\![\_]\!]$ of the form:

$$\begin{aligned} \mathfrak{compile}[\![e]\!] = \ & \texttt{INSTRUCTION}_1 \\ & \vdots \\ & \texttt{INSTRUCTION}_n \end{aligned} \tag{20}$$

The `Constant` expression 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:

  $$[\![\mathfrak{Int}]\!] \stackrel{\text{def}}{=} \mathbb{Z} = \{\ldots, [\![-1]\!], [\![0]\!], [\![1]\!], \ldots\}$$

  $$[\![\mathfrak{Real}]\!] \stackrel{\text{def}}{=} \mathbb{R} = \{\ldots, [\![-1.23]\!], \ldots, [\![0.0]\!], \ldots, [\![1.23]\!], \ldots\}$$

  $$[\![\mathfrak{Char}]\!] \stackrel{\text{def}}{=} \text{set of all Unicode characters}$$

  $$[\![\mathfrak{String}]\!] \stackrel{\text{def}}{=} \text{set of all finite strings of Unicode characters.}$$

  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:

$$\begin{array}{rll}
\texttt{[void]} & \dfrac{}{\Gamma \vdash \mathfrak{void} : \mathfrak{Void}} & \\[2ex]
\texttt{[true]} & \dfrac{}{\Gamma \vdash \mathfrak{true} : \mathfrak{Boolean}} & \\[2ex]
\texttt{[false]} & \dfrac{}{\Gamma \vdash \mathfrak{false} : \mathfrak{Boolean}} & \\[2ex]
\texttt{[int]} & \dfrac{}{\Gamma \vdash n : \mathfrak{Int}} & \textbf{\textit{if}}\ n \text{ is an integer} \\[2ex]
\texttt{[real]} & \dfrac{}{\Gamma \vdash n : \mathfrak{Real}} & \textbf{\textit{if}}\ n \text{ is a floating-point number} \\[2ex]
\texttt{[char]} & \dfrac{}{\Gamma \vdash c : \mathfrak{Char}} & \textbf{\textit{if}}\ c \text{ is a character} \\[2ex]
\texttt{[string]} & \dfrac{}{\Gamma \vdash s : \mathfrak{String}} & \textbf{\textit{if}}\ s \text{ is a string}
\end{array} \tag{21}$$

– COMPILING RULE Compiling a constant consists in pushing the value it denotes on the stack of corresponding sort.

$$
\begin{aligned}
&\texttt{[void]}\ \mathit{compile}[\![\mathfrak{void}]\!] = \textsc{No\_Op}\\
&\texttt{[true]}\ \mathit{compile}[\![\mathfrak{true}]\!] = \textsc{Push\_True}\\
&\texttt{[false]}\ \mathit{compile}[\![\mathfrak{false}]\!] = \textsc{Push\_False}\\
&\texttt{[int]}\ \mathit{compile}[\![n]\!] = \textsc{Push\_I}\ n \quad \textbf{\textit{if}}\ n \text{ is an integer}\\
&\texttt{[real]}\ \mathit{compile}[\![n]\!] = \textsc{Push\_R}\ n \quad \textbf{\textit{if}}\ n \text{ is a floating-point number}\\
&\texttt{[char]}\ \mathit{compile}[\![c]\!] = \textsc{Push\_I}\ c \quad \textbf{\textit{if}}\ c \text{ is a character}\\
&\texttt{[string]}\ \mathit{compile}[\![s]\!] = \textsc{Push\_O}\ s \quad \textbf{\textit{if}}\ s \text{ is a string}
\end{aligned}
\tag{22}
$$

The `Abstraction` *expression*

– 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:

$$\mathfrak{fun}\ x_1, \ldots, x_n\ \cdot\ e \tag{23}$$

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. Other (non-exitable) abstractions are those that are implicitly generated by syntactic desugaring of surface syntax. 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.

$$\frac{\Gamma[x_1 : T_1] \cdots [x_n : T_n] \vdash e : T}{\Gamma \vdash \mathfrak{fun}\ x_1, \ldots, x_n\ \cdot\ e : T_1, \ldots, T_n \to T}\ \textbf{\textit{if}}\ \begin{array}{l}\mathfrak{fun}\ x_1, \ldots, x_n\ \cdot\ e\\ \text{is not exitable.}\end{array} \tag{24}$$

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

$$\frac{\Gamma_{\aleph \leftarrow a}[x_1 : T_1] \cdots [x_n : T_n] \vdash e : T}{\Gamma \vdash a : T_1, \ldots, T_n \to T}\ \textbf{\textit{if}}\ a \text{ is exitable} \tag{25}$$

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

– COMPILING RULE Compiling an abtraction consists in compiling a flattened version of its body (uncurrying and computing parameters offsets), and then generating an instruction pushing a closure on the stack.

$$\begin{array}{l}\mathit{compile}[\![\mathtt{fun}\ x_1,\ldots,x_n\ \cdot\ e]\!] = \mathit{compile}[\![(\mathit{flatten}(e), \mathit{offsets}(x_1,\ldots,x_n)]\!] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{Push\_Closure}\end{array} \tag{26}$$

The `Application` expression

– ABSTRACT SYNTAX This is the familiar function call:

$$f(e_1,\ldots,e_n) \tag{27}$$

– TYPING RULE The type rule is as expected, modulo all potential un/currying that may be needed:

$$\frac{\Gamma \vdash e_1 : T_1,\ \cdots,\ \Gamma \vdash e_n : T_n,\quad \Gamma \vdash f : T_1,\ldots,T_n \to T}{\Gamma \vdash f(e_1,\ldots,e_n) : T} \tag{28}$$

– COMPILING RULE

$$\begin{array}{l}\mathit{compile}[\![f(e_1,\ldots,e_n)]\!] = \mathit{compile}[\![e_n]\!] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{compile}[\![e_1]\!] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{compile}[\![f]\!] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{Apply}\end{array} \tag{29}$$

The `IfThenElse` expression

– ABSTRACT SYNTAX This is the familiar conditional:

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

– TYPING RULE

$$\frac{\Gamma \vdash c : \mathfrak{Boolean},\ \Gamma \vdash e_1 : T,\ \Gamma \vdash e_2 : T}{\Gamma \vdash \mathtt{if}\ c\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T} \tag{30}$$

– COMPILING RULE

$$\begin{array}{l}\mathit{compile}[\![\mathtt{if}\ c\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\!] = \qquad \mathit{compile}[\![c]\!] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{Jump\_On\_False}\ \mathit{jof} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{compile}[\![e_1]\!] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{Jump}\ \mathit{jmp} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{jof} : \mathit{compile}[\![e_2]\!] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{jmp} : \ldots\end{array} \tag{31}$$

The `AndOr` expression

– ABSTRACT SYNTAX

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

– TYPING RULE

$$\frac{\Gamma \vdash e_1 : \mathfrak{Boolean}, \quad \Gamma \vdash e_2 : \mathfrak{Boolean}}{\Gamma \vdash e_1 \ \mathfrak{and}/\mathfrak{or} \ e_2 : \mathfrak{Boolean}} \tag{32}$$

– COMPILING RULE

$$\begin{aligned}
\mathfrak{compile}[\![e_1 \ \mathfrak{and} \ e_2]\!] = \quad & \mathfrak{compile}[\![e_1]\!] \\
& \textsc{Jump_On_False} \ jof \\
& \mathfrak{compile}[\![e_2]\!] \\
& \textsc{Jump_On_True} \ jot \\
jof : \ & \textsc{Push_False} \\
& \textsc{Jump} \ jmp \\
jot : \ & \textsc{Push_True} \\
jmp : \ & \ldots
\end{aligned} \tag{33}$$

$$\begin{aligned}
\mathfrak{compile}[\![e_1 \ \mathfrak{or} \ e_2]\!] = \quad & \mathfrak{compile}[\![e_1]\!] \\
& \textsc{Jump_On_True} \ jot \\
& \mathfrak{compile}[\![e_2]\!] \\
& \textsc{Jump_On_False} \ jof \\
jot : \ & \textsc{Push_True} \\
& \textsc{Jump} \ jmp \\
jof : \ & \textsc{Push_False} \\
jmp : \ & \ldots
\end{aligned} \tag{34}$$

The `Sequence` expression

– ABSTRACT SYNTAX

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

– TYPING RULE

$$\frac{\Gamma \vdash e_1 : T_1, \quad \ldots, \quad \Gamma \vdash e_n : T_n}{\Gamma \vdash \{ \ e_1; \ \ldots; e_n \ \} : T_n} \tag{35}$$

– COMPILING RULE

$$\begin{aligned}
\mathfrak{compile}[\![\{ \ e_1; \ \ldots; e_n \ \}]\!] = \quad & \mathfrak{compile}[\![e_1]\!] \\
& \textsc{Pop_}sort(e_1) \\
& \vdots \\
& \mathfrak{compile}[\![e_n]\!]
\end{aligned} \tag{36}$$

The `WhileDo` expression

– ABSTRACT SYNTAX

$$\mathfrak{while}\ c\ \mathfrak{do}\ e \tag{37}$$

where c and e are expressions.

– TYPING RULE

$$\frac{\Gamma \vdash c : \mathfrak{Boolean},\ \ \Gamma \vdash e : T}{\Gamma \vdash \mathfrak{while}\ c\ \mathfrak{do}\ e\ :\ \mathfrak{Void}} \tag{38}$$

– COMPILING RULE

$$\begin{aligned} \mathfrak{compile}[\![\mathfrak{while}\ c\ \mathfrak{do}\ e]\!] = \mathit{loop} :\ & \mathfrak{compile}[\![c]\!] \\ & \textsc{Jump\_On\_False}\ \mathit{jof} \\ & \mathfrak{compile}[\![e]\!] \\ & \textsc{Jump}\ \mathit{loop} \\ \mathit{jof} :\ & \end{aligned} \tag{39}$$

The `ExitWithValue` expression This is a primitive for so-called non-local exit, and may be used to express more complicated control structures such as exception handling.

– ABSTRACT SYNTAX

$$\mathfrak{exit\ with}\ v \tag{40}$$

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;[30]
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.

Then, control follows up with the next instruction.

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 $\mathfrak{exit\ with}\ v$ 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.

[30] Where *stack* here means "stack of *appropriate* runtime sort;" appropppriate, that is, as per the instruction's runtime sort—*viz.*, ending in `_I` for INT, `_R` for REAL, or `_O` for 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 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:

$$\frac{\Gamma \vdash \aleph_\Gamma : T' \to T,\quad \Gamma \vdash v : T}{\Gamma \vdash \mathfrak{exit\ with}\ v : T} \tag{41}$$

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).[31]
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 non-local return instructions NL_RETURN (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

$$\begin{aligned} \mathfrak{compile}[\![\mathfrak{exit\ with}\ v]\!] = \ & \mathfrak{compile}[\![v]\!] \\ & \texttt{NL\_RETURN\_}\mathit{sort}(v) \end{aligned} \tag{42}$$

C Monoids

In this section, all notions and notations relating to monoids as they are used in this paper are recalled and justified.

Mathematically, a monoid is a non-empty set equipped with an associative internal binary operation and an identity element for this operation. Formally, let S be a set, $\star$ a function from $S \times S$ to S, and $\epsilon \in S$; then, $\langle S, \star, \epsilon \rangle$ is a monoid iff, for any x, y, z in S:

$$x \star (y \star z) = (x \star y) \star z \tag{43}$$

[31] This is why Typing Rule (25) needs to treat both kinds of abstractions.

and

$$x \star \epsilon = \epsilon \star x = \epsilon. \tag{44}$$

Most familiar mathematical binary operations define monoids. For example, taking the set of natural numbers $\mathbb{N}$, and the set of boolean values $\mathbb{B} = \{\mathfrak{true}, \mathfrak{false}\}$, the following are monoids:

- $\langle \mathbb{N}, +, 0 \rangle$,
- $\langle \mathbb{N}, *, 1 \rangle$,
- $\langle \mathbb{N}, \max, 0 \rangle$,
- $\langle \mathbb{B}, \vee, \mathfrak{false} \rangle$,
- $\langle \mathbb{B}, \wedge, \mathfrak{true} \rangle$.

The operations of these monoids are so familiar that they need not be explicated. For us, they have a "built -in" semantics that allows us to compute with them since primary school. Indeed, we shall refer to such readily interpreted monoids as *primitive monoids*.[32]

Note that the definition of a monoid does not preclude additional algebraic structure. Such structure may be specified by other equations augmenting the basic monoid equational theory given by the conjunction of equations (43) and (44). For example, all five monoids listed above are *commutative*; namely, they also obey equation (45):

$$x \star y = y \star x \tag{45}$$

for any x, y. In addition, the three last ones (*i.e.*, $\max$, $\vee$, and $\wedge$) are also *idempotent*; namely, they also obey equation (46):

$$x \star x = x \tag{46}$$

for any x.

Not all monoids are primitive monoids. That is, one may define a monoid purely syntactically whose operation only builds a syntactic structure rather than being interpreted using some semantic computation. For example, linear lists have such a structure: the operation is list concatenation and builds a list out of two lists; its identity element is the empty list. A syntactic monoid may also have additional algebraic structure. For example, the monoid of bags is also defined as a commutative syntactic monoid with the disjunct union operation and the empty bag as identity. Or, the monoid of sets is a commutative and idempotent syntactic monoid with the union operation and the empty set as identity.

Because it is not interpreted, a syntactic monoid poses a problem as far as the representation of its elements is concerned. To illustrate this, let us consider an empty-theory algebraic structure; that is, one without any equations—not even associativity nor identity. Let us take such a structure with one binary operation $\star$ on, say, the natural numbers

[32] We call these monoids "primitive" following the presentation of Fegaras and Maier [22] as it adheres to a more operational (as opposed to mathematical) approach more suitable to computer-scientists. Mathematically, however, these should be called "semantic" monoids since they are interpreted by the computation semantics of their operations. See Appendix Section E.1 for an overview of this formalism.

$\mathbb{N}$. Saying that $\star$ is a "syntactic" operation means that it constructs a syntactic term (*i.e.*, an expression tree) by composing two other syntactic terms. We thus can define the set $T_\star$ of $\star$-terms on some base set, say the natural numbers, inductively as the limit $\cup_{n\geq 0} T_n$ where,

$$T_n \stackrel{\text{def}}{=} \begin{cases} \mathbb{N} & \text{if } n = 0 \\ \\ T_{n-1} \cup \{t_1 \star t_2 \mid t_i \in T_k, i = 1, 2, k < n\} & \text{if } n > 0. \end{cases} \tag{47}$$

Clearly, the set $T_\star$ is well defined and so is the $\star$ operation over it. Indeed, $\star$ is a *bona fide* function from $T_\star \times T_\star$ to $T_\star$ mapping two terms t_1 and t_2 in $T_\star$ into a unique term in $T_\star$—namely, $t_1 \star t_2$. This is why $T_\star$ is called the *syntactic* algebra.[33]

Let us now assume that the $\star$ operation is associative—*i.e.*, that $\star$-terms verify Equation (43). Note that this equation defines a (syntactic) *congruence* on $T_\star$ which identifies terms such as, say, $1 \star (2 \star 3)$ and $(1 \star 2) \star 3$. In fact, for such an associative $\star$ operation, the set $T_\star$ defined in Equation (47) is not the appropriate domain. Rather, the right domain is the quotient set whose elements are (syntactic) congruence classes modulo associativity of $\star$. Therefore, this creates an ambiguity of representation of the syntactic structures.[34]

Similarly, more algebraic structure defined by larger equational theories induces coarser quotients of the empty-theory algebra by putting together in common congruence classes all the syntactic expressions that can be identified modulo the theory's equations. The more equations, the more ambiguous the syntactic structures of expressions. Mathematically, this poses no problem as one can always abstract away from individuals to congruence classes. However, operationally one must resort to some concrete artifact to obtain a unique representation for all members of the same congruence class. One way is to devise a *canonical* representation into which to transform all terms. For example, an associative operation could systematically "move" nested subtrees from its left argument to its right argument—in effect using Equation (43) as a one-way rewrite rule. However, while this is possible for some equational theories, it is not so in general—*e.g.*, take commutativity.[35]

From a programming standpoint (which is ours), we can abstract away from the ambiguity of canonical representations of syntactic monoid terms using a flat notation. For example, in LISP and Prolog, a list is seen as the (flat) sequence of its constituents.

---

[33] For a fixed set of base elements and operations (which constitute what is formally called a *signature*), the syntactic algebra is unique (up to isomorphism). This algebra is also called the *free*, or the *initial*, algebra for its signature.

[34] Note that this ambiguity never arises for semantic algebras whose operations are interpreted into a unique result.

[35] Such are important considerations in the field of *term rewriting* [20], where the problem of finding canonical term representations for equational theories was originally addressed by Donald Knuth and Peter Bendix in a seminal paper proposing a general effective method—the so-called Knuth-Bendix Completion Algorithm [29]. The problem, incidentally, is only semi-decidable. In other words, the Knuth-Bendix algorithm may diverge, although several interesting variations have been proposed for a wide extent of practical uses (see [20] for a good introduction and bibliography).

Typically, a programmer writes $[1, 2, 1]$ to represent the list whose elements are 1, 2 and 1 in this order, and does not care (nor need s/he be aware) of its concrete representation. A set—*i.e.*, a commutative idempotent syntactic monoid—is usually denoted by the usual mathematical notation $\{1, 2\}$, implicitly relying on disallowing duplicate elements, not minding the order in which the elements appear. A bag, or multiset—*i.e.*, a commutative but non-idempotent syntactic monoid—uses a similar notation, allowing duplicate elements but paying no heed to the order in wich they appear; *i.e.*, $\{\!\{1, 2, 1\}\!\}$ is the bag containing 1 twice, and 2 once.

Syntactic monoids are quite useful for programming as they provide adequate data structures to represent collections of objects of a given type. Thus, we refer to them as *collection monoids*. Now, a definition such as Equation (47) for a syntactic monoid, although sound mathematically, is not quite adequate for programming purposes. This is because it defines the $\star$ operations on two distinct *types* of elements; namely, the base elements (here natural numbers) and constructed elements. In programming, it is desirable that operations be given a crisp type. A way to achieve this is by systematically "wrapping" each base element x into a term such as $x \star \epsilon$. This "wrapping" is done by associating to the monoid a function $\mathfrak{U}_\star$ from the base set into the monoid domain called its *unit injection*. For example, if $+\!\!+$ is the list monoid operation for concatenating two lists, $\mathfrak{U}_{+\!\!+}(x) = [x]$ and one may view the list $[a, b, c]$ as $[a]+\!\!+[b]+\!\!+[c]$. Similarly, the set $\{a, b, c\}$ is viewed as $\{a\} \cup \{b\} \cup \{c\}$, and the bag $\{\!\{a, b, c\}\!\}$ as $\{\!\{a\}\!\} \uplus \{\!\{b\}\!\} \uplus \{\!\{c\}\!\}$. Clearly, this bases the constructions on an isomorphic view of the base set rather than the base set itself, while using a uniform type for the monoid operator. Also, because the type of the base elements is irrelevant for the construction other than imposing the constraint that all such elements be of the same type, we present a collection monoid as a *polymorphic* data type. This justifies the formal view of monoids we give next using the programming notion of *polymorphic* type.

Because it is characterized by its operation $\oplus$, a monoid is often simply referred to as $\oplus$. Thus, a monoid operation is used as a subscript to denote its characteristic attributes. Namely, for a monoid $\oplus$,

- $\mathfrak{T}_\oplus$ is its type (*i.e.*, $\oplus : \mathfrak{T}_\oplus \times \mathfrak{T}_\oplus \to \mathfrak{T}_\oplus$),
- $\mathbb{1}_\oplus : \mathfrak{T}_\oplus$ is its identity element,
- $\Theta_\oplus$ is its equational theory (*i.e.*, a subset of the set $\{C, I\}$, where C stands for "commutative" and I for "idempotent");

and, if it is a collection monoid,

- $\mathfrak{C}_\oplus$ is its type constructor (*i.e.*, $\mathfrak{T}_\oplus = \mathfrak{C}_\oplus(\alpha)$),
- $\mathfrak{U}_\oplus : \alpha \to \mathfrak{C}_\oplus(\alpha)$ is its unit injection for any type variable α.

Examples of familiar monoids of both kinds are given in Table 2 in terms of the above characteristic attributes.[36]

[36] If the theory is $\{I\}$—*i.e.*, idempotent but not commutative—this defines yet another, though unfamiliar, type of collection monoid where there may be redundant elements but only if not adjacent.

$\oplus$	$\mathfrak{T}_\oplus$	$\mathbb{1}_\oplus$	$\Theta_\oplus$
$+$	$\mathfrak{Int}$	0	$\{C\}$
$*$	$\mathfrak{Int}$	1	$\{C\}$
$\max$	$\mathfrak{Int}$	0	$\{C, I\}$
$\vee$	$\mathfrak{Boolean}$	$\mathfrak{false}$	$\{C, I\}$
$\wedge$	$\mathfrak{Boolean}$	$\mathfrak{true}$	$\{C, I\}$

Some primitive monoids

$\oplus$	$\mathfrak{C}_\oplus$	$\mathfrak{T}_\oplus$	$\mathbb{1}_\oplus$	$\mathfrak{U}_\oplus(x)$	$\Theta_\oplus$
$\cup$	`set`	$\texttt{set}(\alpha)$	$\{\}$	$\{x\}$	$\{C, I\}$
$\uplus$	`bag`	$\texttt{bag}(\alpha)$	$\{\!\{\}\!\}$	$\{\!\{x\}\!\}$	$\{C\}$
$+\!\!+$	`list`	$\texttt{list}(\alpha)$	$[]$	$[x]$	$\emptyset$

Familiar Collection monoids

Table 2. Attributes of a few common monoids

D The Typed Polymorphic λ-Calculus

We assume a set $\mathcal{C}$ of pregiven constants ususally denoted by $a, b \ldots$, and a countably infinite set of variable symbols $\mathcal{V}$ usually denoted by $x, y, \ldots$. The syntax of a term expression e of the λ-Calculus is given by the grammar shown in Fig. 5. We shall call

$$\begin{array}{rlll} e ::= & a & (a \in \mathcal{C}) & \text{constant} \\ \mid & x & (x \in \mathcal{V}) & \text{variable} \\ \mid & \lambda x.\, e & (x \in \mathcal{V}) & \text{abstraction} \\ \mid & e\ e & & \text{application} \end{array}$$

Fig. 5. Basic λ-Calculus Expressions

$\mathfrak{T}_\Sigma$ the set of term expressions e defined by this grammar. These terms are also called *raw* term expressions.

An abstraction $\lambda x.\, e$ defines a *lexical scope* for its *bound variable* x, whose extent is its *body* e. Thus, the notion of free occurrence of a variable in a term is defined as usual, and so is the operation $e_1[x \leftarrow e_2]$ of substituting a term e_2 for all the free occurrences of a variable x in a term e_1. Thus, a bound variable may be renamed to a new one in its scope without changing the abstraction.

The computation rule defined on λ-terms is the so-called β-reduction:

$$(\lambda x.\, e_1)\ e_2 \ \longrightarrow\ e_1[x \leftarrow e_2]. \tag{48}$$

We assume a set $\mathcal{B}$ of basic type symbols denoted by $A, B, \ldots$, and a countably infinite set of type variables $\mathcal{TV}$ denoted by $\alpha, \beta, \ldots$. The syntax of a type τ of the Typed Polymorphic λ-Calculus is given by the following grammar:

$$\begin{array}{rlll} \tau ::= & A & (A \in \mathcal{B}) & \text{basic type} \\ \mid & \alpha & (\alpha \in \mathcal{TV}) & \text{type variable} \\ \mid & \tau \to \tau & & \text{function type} \end{array} \tag{49}$$

We shall call $\mathfrak{T}$ the set of types τ defined by this grammar. A *monomorphic type* is a type that contains no variable types. Any type containing at least one variable type is called a *polymorphic type*.

The typing rules for the Typed Polymorphic λ-Calculus are given in Fig. 6. These

$$\frac{}{\Gamma \vdash a : \tau} \quad \texttt{if} \quad \mathbf{type}(a) = \tau, \text{ for any type environment } \Gamma \qquad \text{constant}$$

$$\frac{}{\Gamma \vdash x : \tau} \quad \texttt{if} \quad \Gamma(x) = \tau \qquad \text{variable}$$

$$\frac{\Gamma[x : \tau_1] \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \qquad \text{abstraction}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2,\ \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2} \qquad \text{application}$$

Fig. 6. Typing rules for the typed polymorphic λ-calculus

rules can be readily translated into a Logic Programming language based on Horn-clauses such as Prolog, and used as an effective means to infer the types of expressions based on the Typed Polymorphic λ-Calculus.

The basic syntax of the Typed Polymorphic λ-Calculus may be extended with other operators and convenient data structures as long as typing rules for the new constructs are provided. Typically, one provides at least the set $\mathbb{N}$ of integer constants and $\mathbb{B} = \{\mathfrak{true}, \mathfrak{false}\}$ of boolean constants, along with basic arithmetic and boolean operators, pairing (or tupling), a conditional operator, and a fix-point operator. The usual arithmetic and boolean operators are denoted by constant symbols (*e.g.*, $+, *, -, /, \vee, \wedge$, *etc.*). Let $\mathbf{O}$ be this set.

The computation rules for these operators are based on their usual semantics as one might expect, modulo transforming the usual binary infix notation to a "curryed" application. For example, $e_1 + e_2$ is implicitly taken to be the *application* $(+\ e_1)\ e_2$. Note that this means that all such operators are implicitly "curryed."[37]

For example, we may augment the grammar for the terms given in Fig. 5 with the additional rules in Fig. 7.

[37] Recall that a curryed form of an n-ary function f is obtained when f is applied to less arguments than it expects; *i.e.*, $f(e_1, \ldots, e_k)$, for $1 \leq k < n$. In the λ-calculus, this form is simply interpreted as the *abstraction* $\lambda x_1.\ \ldots \lambda x_{n-k}.\ f(e_1, \ldots, e_k, x_1, \ldots, x_{n-k})$. In their fully curried form, all n-ary functions can be seen as unary functions; indeed, with this interpretation of curried forms, it is clear that $f(e_1, \ldots, e_n) = (\ldots (f\ e_1) \ldots e_{n-1})\ e_n$.

$$
\begin{array}{lll}
e ::= \ldots & & \lambda\text{-calculus expression} \\
\quad \mid \langle e, \cdots, e \rangle & & \text{tupling} \\
\quad \mid e.n & (n \in \mathbb{N}) & \text{projection} \\
\quad \mid \mathfrak{if}\ e\ \mathfrak{then}\ e\ \mathfrak{else}\ e & & \text{conditional} \\
\quad \mid \mathfrak{fix}\ e & & \text{fixpoint}
\end{array}
$$

Fig. 7. Additional syntax for the extended λ-calculus (with Fig. 5)

The computation rules for the other new constructs are:

$$
\begin{array}{rcl}
\langle e_1, \cdots, e_k \rangle.i & \longrightarrow & \begin{cases} e_i & \text{if } 1 \leq i \leq k \\ \text{undefined} & \text{otherwise} \end{cases} \\
\mathfrak{if}\ e\ \mathfrak{then}\ e_1\ \mathfrak{else}\ e_2 & \longrightarrow & \begin{cases} e_1 & \text{if } e = \mathfrak{true} \\ e_2 & \text{if } e = \mathfrak{false} \\ \text{undefined} & \text{otherwise} \end{cases} \\
\mathfrak{fix}\ e & \longrightarrow & e\ (\mathfrak{fix}\ e)
\end{array}
\tag{50}
$$

To account for the new constructs, the syntax of types is extended accordingly to:

$$
\begin{array}{lll}
\tau ::= \mathfrak{Int} \mid \mathfrak{Boolean} & & \text{basic type} \\
\quad \mid \alpha & (\alpha \in \mathcal{TV}) & \text{type variable} \\
\quad \mid \langle \tau, \cdots, \tau \rangle & & \text{tuple type} \\
\quad \mid \tau \to \tau & & \text{function type}
\end{array}
\tag{51}
$$

We are given that $\mathbf{type}(n) = \mathfrak{Int}$ for all $n \in \mathbb{N}$ and that $\mathbf{type}(\mathfrak{true}) = \mathfrak{Boolean}$ and $\mathbf{type}(\mathfrak{false}) = \mathfrak{Boolean}$. The (fully curried) types of the built-in operators are given similarly; namely, integer addition has type $\mathbf{type}(+) = \mathfrak{Int} \to (\mathfrak{Int} \to \mathfrak{Int})$, Boolean disjuction has type $\mathbf{type}(\vee) = \mathfrak{Boolean} \to (\mathfrak{Boolean} \to \mathfrak{Boolean})$, *etc.*, ... The additional typing rules for this extended calculus are given in Fig. 8.

E Object Query Language Formalisms

In this section, I review a formal syntax for processiong collections due to Peter Buneman *et al.* [14,15] and elaborated by Leonidas Fegaras and David Maier [22] using the notion of *Monoid Comprehensions*.

E.1 Monoid homomorphisms and comprehensions

The formalism presented here is based on [22] and assumes familiarity with the notions and notations summarized in Appendix Section C. I will use the programming view of monoids exposed there using the specific notation of monoid attributes, in particular for sets, bags, and lists. I will also assume basic familiarity with naive λ-calculus and associated typing as presented in Appendix Section D.

$$\frac{\Gamma \vdash t_1 : \tau_1, \cdots \Gamma \vdash t_k : \tau_k}{\Gamma \vdash \langle t_1, \cdots, t_k \rangle : \langle \tau_1, \cdots, \tau_k \rangle} \quad \text{tupling}$$

$$\frac{\Gamma \vdash t : \langle \tau_1, \cdots, \tau_k \rangle}{\Gamma \vdash t.i : \tau_i} \ \texttt{if}\ 1 \leq i \leq k \quad \text{tuple projection}$$

$$\frac{\Gamma \vdash t_1 : \mathfrak{Boolean}, \Gamma \vdash t_2 : \tau, \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathfrak{if}\ t_1\ \mathfrak{then}\ t_2\ \mathfrak{else}\ t_3 : \tau} \quad \text{conditional}$$

$$\frac{\Gamma \vdash t : \tau \to \tau}{\Gamma \vdash \mathfrak{fix}\ t : \tau} \quad \text{fixpoint}$$

Fig. 8. Additional typing rules for the extended typed polymorphic λ-calculus (with Fig. 6)

Monoid homomorphisms Because many operations and data structures are monoids, it is interesting to use the associated concepts as the computational building block of an essential calculus. In particular, iteration over collection types can be elegantly formulated as computing a *monoid homomorphism*. This notion coincides with the usual mathematical notion of homomorphism, albeit given here from an operational standpoint and biased toward collection monoids. Basically, a monoid homomorphism $\mathfrak{hom}^{\odot}_{\oplus}$ maps a function f from a collection monoid $\oplus$ to *any* monoid $\odot$ by collecting all the f-images of elements of a $\oplus$-collection using the $\odot$ operation. For example, the expression $\mathfrak{hom}^{\cup}_{++}[\lambda x.\, x+1]$ applied to the list $[1,2,1,3,2]$ returns the set $\{2,3,4\}$.[38]

In other words, the monoid homomorphism $\mathfrak{hom}^{\cup}_{++}$ of a function f applied to a list L corresponds to the following *loop* computation collecting the f-images of the list elements into a set (each f-image being a set):

```
result ← {};
foreach x in L do result ← result ∪ f(x);
return result;
```

This is formalized as follows:

Definition 1 (Monoid Homomorphism). *A Monoid Homomorphism $\mathfrak{hom}^{\odot}_{\oplus}$ defines a mapping from a collection homomorphism $\oplus$ to any monoid $\odot$ such that $\Theta_{\oplus} \subseteq \Theta_{\odot}$ by:*

$$\begin{aligned}
\mathfrak{hom}^{\odot}_{\oplus}[f](\mathbb{1}_{\oplus}) &\stackrel{def}{=} \mathbb{1}_{\odot} \\
\mathfrak{hom}^{\odot}_{\oplus}[f](\mathfrak{U}_{\oplus}(x)) &\stackrel{def}{=} f(x) \\
\mathfrak{hom}^{\odot}_{\oplus}[f](x \oplus y) &\stackrel{def}{=} \mathfrak{hom}^{\odot}_{\oplus}[f](x) \odot \mathfrak{hom}^{\odot}_{\oplus}[f](y)
\end{aligned}$$

for any function $f : \alpha \to \mathfrak{T}_{\odot}$, $x : \alpha$, and $y : \alpha$, where $\mathfrak{T}_{\oplus} = \mathfrak{C}_{\oplus}(\alpha)$.

Again, computationally, this amounts to executing the following iteration:

```
result ← 𝟙⊙;
foreach x_i in 𝔘⊕(x_1) ⊕ ··· ⊕ 𝔘⊕(x_n) do result ← result ⊙ f(x_i);
return result;
```

[38] See Table 2 for notation of a few common monoids.

The reader may be puzzled by the condition $\Theta_\oplus \subseteq \Theta_\odot$ in Definition 1. It means that a monoid homomorphism may only be defined from a collection monoid to a monoid that has at least the same equational theory. In other words, one can only go from an empty theory monoid, to either a $\{C\}$-monoid or an $\{I\}$-monoid, or yet to a $\{C, I\}$-monoid. This requirement is due to an algebraic technicality, and relaxing it would cause a monoid homomorphism to be ill-defined. To see this, consider going from, say, a commutative-idempotent monoid to one that is commutative but not idempotent. Let us take, for example, $\mathfrak{hom}^+_\cup$. Then, this entails:

$$\begin{aligned}
1 &= \mathfrak{hom}^+_\cup[\lambda x.\,1](\{a\}) \\
&= \mathfrak{hom}^+_\cup[\lambda x.\,1](\{a\} \cup \{a\}) \\
&= \mathfrak{hom}^+_\cup[\lambda x.\,1](\{a\}) \;+\; \mathfrak{hom}^+_\cup[\lambda x.\,1](\{a\}) \\
&= 1 + 1 \\
&= 2.
\end{aligned}$$

The reader may have noticed that this restriction has the unfortunate consequence of disallowing potentially useful computations, notable examples being computing the cardinality of a set, or converting a set into a list. However, this drawback can be easily overcome with a suitable modification of the third clause in Definition 1, and other expressions based on it, ensuring that anomalous cases such as the above are dealt with by appropriate tests.

It is important to note that, for the consistency of Definition 1, a non-idempotent monoid must actually be anti-idempotent, and a non-commutative monoid must be anti-commutative. Indeed, if $\oplus$ is non-idempotent as well as non-anti-idempotent (say, $x_0 \oplus x_0 = x_0$ for some x_0), then this entails:

$$\begin{aligned}
\mathfrak{hom}^\odot_\oplus[f](x_0) &= \mathfrak{hom}^\odot_\oplus[f](x_0 \oplus x_0) \\
&= \mathfrak{hom}^\odot_\oplus[f](x_0) \;\odot\; \mathfrak{hom}^\odot_\oplus[f](x_0)
\end{aligned}$$

which is not necessarily true for non-idempotent $\odot$. A similar argument may be given for commutativity. This consistency condition is in fact not restrictive operationally as it is always verified (*e.g.*, a list will not allow partial commutation of any of its element).

Here are a few familar functions expressed with well-defined monoid homomorphisms:

$$\begin{aligned}
\texttt{length}(l) &= \mathfrak{hom}^+_{+\!+}[\lambda x.\,1](l) \\
e \in s &= \mathfrak{hom}^\vee_\cup[\lambda x.\,x = e](s) \\
s \times t &= \mathfrak{hom}^\cup_\cup[\lambda x.\,\mathfrak{hom}^\cup_\cup[\lambda y.\,\{\langle x, y\rangle\}](t)](s) \\
\texttt{map}(f, s) &= \mathfrak{hom}^\cup_\cup[\lambda x.\,\{f(x)\}](s) \\
\texttt{filter}(p, s) &= \mathfrak{hom}^\cup_\cup[\lambda x.\,\mathfrak{if}\ p(x)\ \mathfrak{then}\ \{x\}\ \mathfrak{else}\ \{\}](s).
\end{aligned}$$

Monoid comprehensions 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 2 (Monoid Comprehension). *The meaning of a monoid comprehension over a monoid $\oplus$ is defined inductively as follows:*

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

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

$$\oplus\{e \parallel c, Q\} \stackrel{def}{=} \mathfrak{if}\ c\ \mathfrak{then}\ \oplus\{e \parallel Q\}\ \mathfrak{else}\ \mathbb{1}_\oplus$$

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.

Note that relational *joins* are immediately expressible as monoid comprehensions. Indeed, the join of two sets S and T using a function f and a predicate p is simply:

$$S \bowtie^f_p T \stackrel{\text{def}}{=} \cup\{f(x,y) \parallel x \leftarrow S, y \leftarrow T, p(x,y)\}. \tag{52}$$

Typically, a relational join will take f to be a record constructor. For example, if we write a record whose fields $\mathtt{l}_i$ have values e_i for $i = 1, \ldots, n$, as $\langle \mathtt{l}_1 = e_1, , \ldots, \mathtt{l}_n = e_n\rangle$, then a standard relational join can be obtained with, say, $f(x,y) = \langle \mathtt{name} = y.\mathtt{name}, \mathtt{age} = 2 * x.\mathtt{age}\rangle$, and $p(x,y)$ may be any condition such as $x.\mathtt{name} = y.\mathtt{name}, x.\mathtt{age} \geq 18$.

Clearly, monoid comprehensions can immediately express queries using all usual relational operators (and, indeed, object queries as well) and most usual functions. For example,

$$
\begin{array}{llll}
\exists x \in s.e & \stackrel{\text{def}}{=} \vee\{e \parallel x \leftarrow s\} & \texttt{length}(s) & \stackrel{\text{def}}{=} +\{1 \parallel x \leftarrow s\} \\
\forall x \in s.e & \stackrel{\text{def}}{=} \wedge\{e \parallel x \leftarrow s\} & \texttt{sum}(s) & \stackrel{\text{def}}{=} +\{x \parallel x \leftarrow s\} \\
x \in s & \stackrel{\text{def}}{=} \vee\{x = y \parallel y \leftarrow s\} & \texttt{max}(s) & \stackrel{\text{def}}{=} \max\{x \parallel x \leftarrow s\} \\
s \cap t & \stackrel{\text{def}}{=} \cup\{x \parallel x \leftarrow s, x \in t\} & \texttt{filter}(p, s) & \stackrel{\text{def}}{=} \cup\{x \parallel x \leftarrow s, p(x)\} \\
\texttt{count}(a, s) & \stackrel{\text{def}}{=} +\{1 \parallel x \leftarrow s, x = a\} & \texttt{flatten}(s) & \stackrel{\text{def}}{=} \cup\{x \parallel t \leftarrow s, x \leftarrow t\}
\end{array}
$$

Note that some of these functions will work only on appropriate types of their arguments. For example, the type of the argument of `sum` must be a non-idempotent monoid, and so must the type of the second argument of `count`. Thus, `sum` will add up the elements of a bag or a list, and `count` will tally the number of occurrences of an element in a bag or a list. Applying either `sum` or `count` to a set will be caught as a type error.

We are now in a position to propose a programming calculus using monoid comprehensions. Fig. 9 defines an abstract grammar for an expression e of the *Monoid Comprehension Calculus* and amounts to adding comprehensions to an extended Typed Polymorphic λ-Calculus. Fig. 10 gives the typing rules for this calculus.

$e ::= \ldots$	extended λ-calculus expression
$\mid \mathbb{1}_{\oplus}$	monoid identity
$\mid \mathfrak{U}_{\oplus}(e)$	monoid unit injection
$\mid e_1 \oplus e_2$	monoid composition
$\mid \oplus\{e \parallel Q\}$	monoid comprehension

Fig. 9. Additional Syntax for the monoid comprehension calculus (with Fig. 7)

F Backend System

Our generic backend system comprises classes for managing runtime events and objects, a display manager, and an error manager. As an example, we describe the organization of a runtime object.

The class $\langle backend \rangle$`.Runtime.java` defines what a runtime context consists of as an object of this class. Such an object serves as the common execution environment context shared by $\langle instructions \rangle$`.Instruction` objects being executed. It encapsulates a state of comptutation that is effected by each instruction as it is executed in its context.

Thus, a $\langle backend \rangle$`.Runtime.java` object consists of attributes and structures that together define a state of computation, and methods that are used by instructions

$$\frac{}{\Gamma \vdash \mathbb{1}_{\oplus} : \mathfrak{T}_{\oplus}} \quad \mathbb{1}_{\oplus} \text{ monoid identity}$$

$$\frac{\Gamma \vdash e_1 : \mathfrak{T}_{\oplus},\ \Gamma \vdash e_2 : \mathfrak{T}_{\oplus}}{\Gamma \vdash e_1 \oplus e_2 : \mathfrak{T}_{\oplus}} \quad \oplus \text{ primitive monoid}$$

$$\frac{\Gamma \vdash e : \mathfrak{T}_{\oplus}}{\Gamma \vdash \oplus\{e \parallel\} : \mathfrak{T}_{\oplus}} \quad \oplus \text{ primitive monoid}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathfrak{U}_{\oplus}(e) : \mathfrak{C}_{\oplus}(\tau)} \quad \oplus \text{ collection monoid}$$

$$\frac{\Gamma \vdash e_1 : \mathfrak{C}_{\oplus}(\tau),\ \Gamma \vdash e_2 : \mathfrak{C}_{\oplus}(\tau)}{\Gamma \vdash e_1 \oplus e_2 : \mathfrak{C}_{\oplus}(\tau)} \quad \oplus \text{ collection monoid}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \oplus\{e \parallel\} : \mathfrak{C}_{\oplus}(\tau)} \quad \oplus \text{ collection monoid}$$

$$\frac{\Gamma \vdash e_2 : \mathfrak{C}_{\odot}(\tau_2),\ \Gamma[x : \tau_2] \vdash \oplus\{e_1 \parallel Q\} : \tau_1}{\Gamma \vdash \oplus\{e_1 \parallel x \leftarrow e_2, Q\} : \tau_1} \ \textit{if}\ \ \Theta_{\odot} \subseteq \Theta_{\oplus} \quad \text{subtheory}$$

$$\frac{\Gamma \vdash e_2 : \mathfrak{Boolean},\ \Gamma \vdash \oplus\{e_1 \parallel Q\} : \tau}{\Gamma \vdash \oplus\{e_1 \parallel e_2, Q\} : \tau}$$

Fig. 10. Additional typing rules for the monoid comprehension calculus (with Fig. 6)

to effect this state as they are executed. Thus, each instruction subclass of ⟨*instructions*⟩`.Instruction` defines an `execute(`⟨*backend*⟩`.Runtime)` method that specifies its operational semantics as a state transformation of its given runtime context.

Initiating execution of a ⟨*backend*⟩`.Runtime.java` 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 and invoking `execute(this)` on whatever instruction in the current code array for this `Runtime.java` object is currently at address `_ip`. 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 [30]—in the same spirit as Luca Cardelli's Functional Abstract Machine (FAM) [16], 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 ⟨*back-end*⟩`.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 ⟨*types*⟩`.TypeChecker` and the ⟨*kernel*⟩`.Sanitizer`, which ascertain all the conditions that must be met prior to having a ⟨*kernel*⟩`.Compiler` proceed to generating instructions which will safely act on the appropriate stacks and environments of the correct sort (integer, floating-point, or object).

Display manager objects and error manager objects are similarly organized.

References

1. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA (USA), 1974.
2. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley, 1986.
3. AÏT-KACI, H. *Warren's Abstract Machine—A Tutorial Reconstruction.* Logic Programming. MIT Press, Cambridge, MA (USA), 1991.
4. AÏT-KACI, H. An introduction to LIFE—Programming with Logic, Inheritance, Functions, and Equations. In *Proceedings of the International Symposium on Logic Programming* (October 1993), D. Miller, Ed., MIT Press, pp. 52–68.
5. AÏT-KACI, H. An Abstract and Reusable Programming Language Architecture. Invited presentation, LDTA'03[39], April 6, 2003. [Available online.[40]].
6. AÏT-KACI, H. A generic XML-generating metacompiler. Part of the documentation of the Jacc package, July 2008. [Available online.[41]].
7. AÏT-KACI, H., AND DI COSMO, R. Compiling order-sorted feature term unification. PRL Technical Note 7, Digital Paris Research Laboratory, Rueil-Malmaison, France, December 1993.
8. BANÂTRE, J.-P., AND LE MÉTAYER, D. A new computational model and its discipline of programming. INRIA Technical Report 566, Institut National de Recherche en Informatique et Automatique, Le Chesnay, France, 1986.
9. BERRY, G., AND BOUDOL, G. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL'90* (New York, NY, USA, 1990), ACM Press, pp. 81–94. [Available online.[42]].
10. BIERMAN, G. M., GORDON, A. D., HRIŢCU, C., AND LANGWORTHY, D. Semantic subtyping with an SMT solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programmingm (ICFP 2010, Baltimore, MA USA)* (New York, NY (USA), September 27–29, 2010), Association for Computing Machinery, ACM, pp. 105–116. [Available online.[43]].

[39] http://ldta.info/2003/

[40] http://hassan-ait-kaci.net/pdf/ldta03.pdf

[41] http://www.hassan-ait-kaci.net/jacc-xml.pdf

[42] citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.3782

[43] http://research.microsoft.com/apps/pubs/?id=135577

11. BIERMAN, G. M., GORDON, A. D., HRIŢCU, C., AND LANGWORTHY, D. Semantic subtyping with an SMT solver. *Journal of Functional Programming* (2012), 1–75. [Available online.[44]]—*N.B.*: Full version of [10].
12. BOTHNER, P. XQuery tutorial. Online tutorial. [Available online.[45]].
13. BRODKY, A., SEGAL, V. E., CHEN, J., AND EXARKHOPOULO, P. A. The CCUBE system object-oriented database system. In *Constraints and Databases*, R. Ramakrishnan and P. J. Stuckey, Eds. Kluwer Academic Publishers, Norwell, MA (USA), 1998, pp. 245–277. (Special Issue on *Constraints: An International Journal*, 2(3&4), 1997.).
14. BUNEMAN, P., LIBKIN, L., SUCIU, D., TANNEN, V., AND WONG, L. Comprehension syntax. *ACM SIGMOD Record 23*, 1 (March 1994), 87–96. [Available online.[46]].
15. BUNEMAN, P., NAQVI, S., TANNEN, V., AND WONG, L. Principles of programming with complex objects and collection types. *Theoretical Computer Science 149*, 1 (January 1995), 3–48. [Available online.[47]].
16. CARDELLI, L. The functional abstract machine. Technical Report TR-107, AT&T Bell Laboratories, Murray Hill, New Jersey, May 1983. [Available online.[48]].
17. CARDELLI, L. Typeful programming. In *Formal Description of Programming Concepts*, E. J. Neuhold and M. Paul, Eds. Springer-Verlag, 1991. [Available online.[49]].
18. CHOE, K.-M. Personal communication. Korean Advanced Institute of Science and Technology, Seoul, South Korea, December 2000. `choe@compiler.kaist.ac.kr`.
19. DEREMER, F., AND PENNELLO, T. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems 4*, 4 (April 1982), 615–649. [Available online.[50]].
20. DERSHOWITZ, N. A taste of rewriting. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, P. E. Lauer, Ed. Springer-Verlag, 1993, pp. 199–228. [Available online.[51]].
21. FEGARAS, L. An experimental optimizer for OQL. Technical Report TR-CSE-97-007, University of Texas at Arlington, May 1997. [Available online.[52]].
22. FEGARAS, L., AND MAIER, D. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems 25*, 4 (December 2000), 457–516. [Available online.[53]].
23. GESBERT, N., GENEVÈS, P., AND LAYAÏDA, N. Parametric polymorphism and semantic subtyping: the logical connection. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming (ICFP 2011, Tokyo Japan)* (New York, NY (USA), September 19–21, 2011), Association for Computing Machinery, ACM, pp. 107–116. [Available online.[54]].
24. GESBERT, N., GENEVÈS, P., AND LAYAÏDA, N. Parametric polymorphism and semantic subtyping: the logical connection. *SIGPLAN Notices 46*, 9 (September 2011). *N.B.*: full version of [23].

[44] http://www-infsec.cs.uni-saarland.de/~hritcu/publications/dminor-jfp2012.pdf
[45] http://www.gnu.org/software/qexo/XQuery-Intro.html
[46] http://www.acm.org/sigs/sigmod/record/issues/9403/Comprehension.ps
[47] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.5516
[48] http://lucacardelli.name/Papers/FAM.pdf
[49] http://lucacardelli.name/Papers/TypefulProg.A4.pdf
[50] http://dl.acm.org/citation.cfm?id=69622.357187
[51] http://www-sal.cs.uiuc.edu/~nachum/papers/taste-fixed.ps.gz
[52] http://lambda.uta.edu/oqlopt.ps.gz
[53] http://lambda.uta.edu/tods00.ps.gz
[54] http://hal.inria.fr/inria-00585686/fr/

25. GRUST, T. Monad comprehensions—a versatile representation for queries. In *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*, P. Gray, L. Kerschberg, P. King, and A. Poulovassilis, Eds. Springer, September 2003. [Available online.[55]].
26. HENTENRYCK, P. *The OPL Optimization Programming Language*. The MIT Press, 1999.
27. JAFFAR, J., AND MAHER, M. J. Constraint Logic Programming: A survey. *Journal of Logic Programming 19/20* (1994), 503–581. [Available online.[56]].
28. JOHNSON, S. Yacc: Yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Labs, Murray Hill, NJ, 1975. (Reprinted in the 4.3BSD Unix Programmer's Manual, Supplementary Documents 1, PS1:15. UC Berkeley, 1986.).
29. KNUTH, D. E., AND BENDIX, P. B. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, J. Leech, Ed. Pergamon Press, Oxford, UK, 1970, pp. 263–297. Reprinted in *Automatic Reasoning*, 2, Springer-Verlag, pp. 342–276 (1983).
30. LANDIN, P. J. The mechanical evaluation of expressions. *Computer Journal 6*, 4 (1963), 308–320. [Available online.[57]].
31. LANDIN, P. J. The next 700 programming languages. *Communications of the ACM 9*, 3 (March 1966), 157–166. [Available online.[58]].
32. LEROY, X. Unboxed objects and polymorphic typing. In *Proceedings of the 19th symposium Principles of Programming Languages (POPL'92)* (1992), Association for Computing Machinary, ACM Press, pp. 177–188. [Available online.[59]].
33. NIC, M., AND JIRAT, J. XPath tutorial. Online tutorial. [Available online.[60]].
34. PARK, J., CHOE, K.-M., , AND CHANG, C. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems 7*, 1 (January 1985), 159–175. [Available online.[61]].
35. PLOTKIN, G. D. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, University of Århus, Århus, Denmark, 1981. [Available online.[62]].
36. PLOTKIN, G. D. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming 60–61* (January 30, 2004), 17–139. [Available online.[63]]—*N.B.*: Published version of [35].
37. SETHI, R. *Programming Languages—Concepts and Constructs*, 2nd ed. Addison-Wesley, Reading, MA (USA), 1996.
38. VISSER, E. *Syntax Definition for Language Prototyping*. PhD thesis, Faculteit Wiskunde, Informatics, Natuurkunde en Strenkunde, Universiteit van Amsterdam, Amsterdam, The Netherlands, September 1997. [Available online.[64]].
39. WONG, L. *Querying Nested Collections*. PhD thesis, University of Pennsylvania (Computer and Information Science), 1994. [Available online.[65]].

[55] http://www-db.in.tum.de/~grust/files/monad-comprehensions.pdf
[56] http://citeseer.ist.psu.edu/jaffar94constraint.html
[57] http://www.cs.cmu.edu/~crary/819-f09/Landin64.pdf
[58] http://www.thecorememory.com/Next_700.pdf
[59] http://gallium.inria.fr/~xleroy/bibrefs/Leroy-unboxed.html
[60] http://www.zvon.org/xxl/XPathTutorial/General/examples.html
[61] http://dl.acm.org/citation.cfm?id=69622.357187
[62] http://citeseer.ist.psu.edu/673965.html
[63] http://homepages.inf.ed.ac.uk/gdp/publications/sos_jlap.pdf
[64] http://eelcovisser.org/wiki/thesis
[65] ftp://ftp.cis.upenn.edu/pub/ircs/tr/94-09.ps.Z