# Extensional filters reveal algorithms*

**Guillaume Bonfante[1], Pierre Boudes[2], and Jean-Yves Moyen[3]**

1 **LORIA – BP239, 54506 Vandœuvre-lès-Nancy Cedex France**
`Guillaume.Bonfante@loria.fr`
2,3 **LIPN – CNRS, Université Paris 13 – 99 Villetaneuse, France**
`{boudes, moyen}@lipn.univ-paris13.fr`

**Abstract**

This contribution deals with the intensionality of programming languages, and especially addresses the question: "*How to compare programming languages from an algorithmic point of view?*" We propose an indirect way to compare two languages computing a same set of functions, based on the use of a witness subset of programs, called a filter. If two languages compute the same set of function before filtering, but two different set after filtering, we argue that the initial languages were *intensionally* different and claim that the difference is in the *algorithms* they implement. Filtering thus provides a tool to perform an intensional observation of languages.

*Implicit Computational Complexity* (ICC) aims at describing complexity classes, that is sets of functions, without reference to some machinery. A description is usually a constrained programming language (via types for instance). A central notion in ICC is the intensionality, of such constrained programming languages. Thus a typical question in ICC is "*Given two languages describing* Ptime*, which one contains more algorithms (or better programs)?*" In other words, a constrained language is more powerful and easier to use if it is more expressive than another.

A race for more expressive characterizations drove the community of ICC, especially describing Ptime. For instance, interpretation methods to prove termination [3] go beyond Jones' characterization of polynomial time [9], but are themselves "overwhelmed" by quasi-interpretations [4].

Immediately, a question arises: "*How can one compare two such descriptions of some class of functions?*" Naturally, if the two classes of functions are distinct, so must be the two initial descriptions. But what happens with two descriptions of the same class of functions ? The problem has been already stated in the past, for instance by Grigorieff and Valarcher in an illuminating discussion about the Abstract State Machines Thesis of Gurevich [7].

In a sense, being able to compute a function is not enough, a language should allow to write the "good" program computing a function, or failing that, various ways to do it (and let the programmer decide which is the good one). Among *extensionnally equivalent* languages, there may be some implementing a broader variety of algorithms than others. Thus, the intensional content of programming languages should be expressed in term of algorithms. But giving a real mathematical meaning to this point of view is hard because the notion of algorithm stays quite subjective.

In this work, we do not try to formally define algorithms. We keep the notion informal. We say that a program implements an algorithm, and an algorithm computes a function. Thus, an algorithm gathers "equivalent" programs. We are aware of the thesis of Blass, Dershowitz and Gurevich who defend [1] that there is no way to see algorithms as equivalence classes of programs. However, we do not see any better word to speak about these equivalence classes.

* This work was supported by the ANR-14-CE25-0005 ELICA

Felleisen proposed [6] to capture the expressive power of programming languages by looking at the translations of programs between languages. If one language has a programming construct for which any translation into another language involve complex reformulations of large fractions of the program then the first language is more expressive thanks to this construct. For example, an exception mechanism requires an heavy translation to be simulated while a `let in` can be simulated by a local inlining.

We propose to work in an indirect way, on the side of sets of computed functions. This approach has been used in the past, for instance by Colson in his remarkable work on primitive recursion [5] where he proves that no algorithm (no primitive recursive function) computes the minimum function of two tally numbers $m$ and $n$ in time $O(\min(m, n))$. We want to develop a more systematic way to discriminate sets of programs.

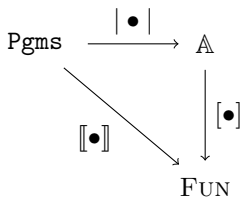
The result is illustrated with classical examples from ICC, to show its effectiveness. It is worth noting that to work, the filtering procedure requires several proofs which functions are computed by a given set of programs. These proofs are usually hard to obtain and this whole work is thus fueled by the knowledge gathered during two decades of ICC.

## 1 Functions, algorithms and programs

Let $1_X$ denotes the identity on $X$. We use $\hookrightarrow$ to denote injections and $\twoheadrightarrow$ to denote surjections. If $X' \subseteq X$, $\iota : X' \hookrightarrow X$ is the canonical injection.

All along, we suppose given a fixed set Fun of functions over some domain $\mathcal{D}$. For example, functions over binary trees as done by Jones in [8]. A priori, Fun is not denumerable. In practice, we suppose that Fun contains all Turing-computable functions. Thus, we can see Ptime, Pspace as subsets of Fun. We also fix a global programming language `Pgms` and its semantics function $[\![-]\!] : \texttt{Pgms} \to \textsc{Fun}$. Example include Turing Machines, $\lambda$-calculus or `C`. The co-domain of $[\![-]\!]$ is the set of *computable functions*.

We suppose furthermore that there is a notion of *algorithms* that decomposes the semantics. That is, there exists a set $\mathbb{A}$ of algorithms, an *implementation* function $|-| : \texttt{Pgms} \to \mathbb{A}$ and a *computation* function $[-] : \mathbb{A} \to \textsc{Fun}$ such that the semantics can be decomposed as $[\![\texttt{p}]\!] = [|\texttt{p}|]$. In other words, the notion of algorithms makes the diagram on the right commute. Note that very little is required on algorithms which may be anything from programs ($|-|$ is identity) to functions ($[-]$ is identity). That is, programmers need to first agree on what an algorithm is before being able to compare languages.

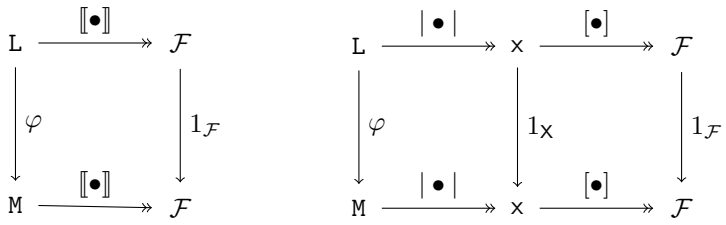



We abusively call "language" any subset of `Pgms`. That is, a language is a set of programs but all languages live within a given universe of programs. Typical examples of languages in the ICC context are the typable $\lambda$-terms, the `CONS`-free programs or the Term Rewriting Systems admitting a quasi-interpretation.

## 2 Filters

### 2.1 Compilation

Let `L` and `M` be two programming languages computing the same set of functions $\mathcal{F}$. A *compilation procedure* is a function $\varphi : \texttt{L} \to \texttt{M}$ which respects the semantics, that is the following diagram on the left-hand-side side commute:



Compilation may optimize programs to the point of changing the implemented algorithm. If we want to compare the expressivity of L and M, we need to disallow this. Thus, we say that two languages are *intensionally equivalent* if there is a compilation procedure that makes the above diagram on the right-hand-side commute. Obviously, this depends on the choice of algorithms, that is $\mathbb{A}$ sets the granularity of the comparison between languages.

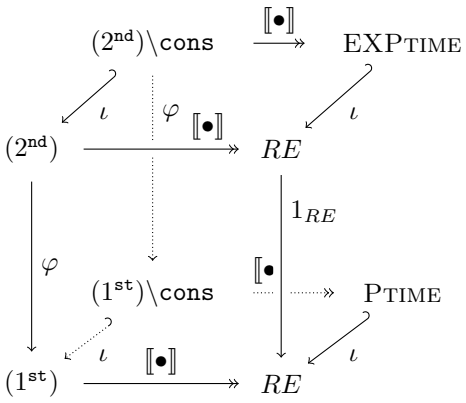
## 2.2 Program filtering

Let us forget for a while the algorithm layer. Suppose that we have two languages L and M. Consider a compilation procedure $\varphi : \mathtt{L} \to \mathtt{M}$ as above. A *filter*[1] is a set of programs $\mathtt{Q} \subseteq \mathtt{Pgms}$. We say that a compilation procedure $\varphi$ *respects a filter* if $\forall \mathtt{p} \in \mathtt{L}, \mathtt{p} \in \mathtt{Q} \Leftrightarrow \varphi(\mathtt{p}) \in \mathtt{Q}$.

▶ **Theorem 1.** *Let L and M be two programming languages computing the same set of functions, that is* $[\![L]\!] = \mathcal{F} = [\![M]\!]$. *If there exists a filter Q such that* $[\![L \bigcap Q]\!] = \mathcal{G} \supsetneq \mathcal{G}' = [\![M \bigcap Q]\!]$ *then there exist no compilation procedure from L to M respecting the filter.*

**Proof.** If such a compilation $\varphi$ exists, let $f \in \mathcal{G}$ and $f \notin \mathcal{G}'$. By definition of $\mathcal{G}$, there is a program p in $\mathtt{L} \bigcap \mathtt{Q}$ such that $[\![\mathtt{p}]\!] = f$. But, then, $\varphi(\mathtt{p}) \in \mathtt{Q}$ since $\varphi$ respects the filter, and $\varphi(\mathtt{p}) \in \mathtt{M}$ by definition, thus $\varphi(\mathtt{p}) \in \mathtt{M} \bigcap \mathtt{Q}$. Hence $[\![\varphi(\mathtt{p})]\!] = f \in [\![\mathtt{M} \bigcap \mathtt{Q}]\!]$ which contradicts the hypothesis. ◀

## 2.3 Example: Life without CONS



We can now express Jones' classical result about the expressive power of high-order [9]. Pgms is the set of deterministic high-order rewriting systems. Let $\mathtt{L} = (2^{\mathtt{nd}})$ be the second order programs and $\mathtt{M} = (1^{\mathtt{st}})$ be the first-order ones. It is well-known that both these languages compute all the recursively enumerable functions, that is $\mathcal{F} = [\![\mathtt{L}]\!] = RE = [\![\mathtt{M}]\!]$ and we can go from high order to first order via *defunctionalisation*.

What can we say about the expressivity of high order compared to the one of first order?

Let $\backslash\mathtt{cons}$ be the set of programs which are cons-free, that is the right-hand side does not contain constructor symbols and write abusively $(1^{\mathtt{st}})\backslash\mathtt{cons}, (2^{\mathtt{nd}})\backslash\mathtt{cons}$ for the intersections. Jones proved that $[\![(1^{\mathtt{st}})\backslash\mathtt{cons}]\!] = [\![\mathtt{M'}]\!] = \textsc{Ptime}$ and $[\![(2^{\mathtt{nd}})\backslash\mathtt{cons}]\!] = [\![\mathtt{L'}]\!] = \textsc{Exptime}$. Thus, there exists no compilation that respects the filter. In other words, any defunctionalisation **must** break this filter, that is add cons to its result.

[1] In the chemical sense, rather than the mathematical one. That is a filter is seen as a device that let goes through only the elements of Q.

Note that the result relies heavily on the previous work of Jones. However, our results goes a slight bit further by specifying a bit the sense of "more expressive", in this case, the absence of any defunctionalisation procedure which respects `cons`-free.

## 2.4 Other examples

It is known that both deterministic and non-deterministic first order rewriting system terminating by multiset path ordering (MPO) compute primitive recursive functions. Does that mean that they are intensionally equivalent and that non-determinism has no power? If we filter by keeping only the systems that admit a Quasi-Interpretation (QI) we get the answer. Deterministic systems terminating by MPO and admitting a QI characterize Ptime while the non-deterministic ones characterize Pspace (which we consider different here). Thus, non-determinism adds expressive power, in this case, there is no determinisation procedure that conserves QI.

Similarly, parameter substitution added to primitive recursion does not allow to compute more function. However, when filtered by predicative recursion, classical primitive recursion is tamed down to Ptime while parameter substitution allow to express Pspace. Again, parameter substitution is more expressive, namely it is not possible to remove it without breaking predicative recursion.

As mentioned, all the results cited here were hard to prove and each one was more or less a breakthrough when first published. That is, our method regroups classical ICC results in a common framework. In this case, the original results are due to Bonfante, Marion, Moyen [4, 2] and Leivant, Marion [10, 11].

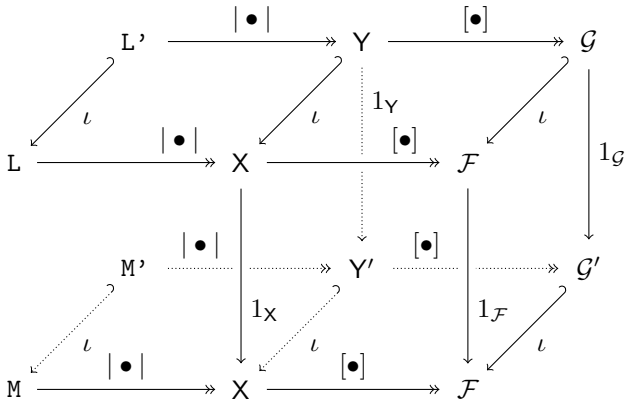
# 3 The extensional filtering discrimination procedure

Let's go back to algorithms. Even without an explicit compilation, we can say that two languages are algorithmically equivalent if they implement the same algorithms.

## 3.1 Extensional filters

An *extensional filter* is a set of programs $\mathtt{Q} \subseteq \mathtt{Pgms}$ such that for all programs $\mathtt{p}$ and $\mathtt{q}$, if $|\mathtt{p}| = |\mathtt{q}|$, then $\mathtt{p} \in \mathtt{Q} \Leftrightarrow \mathtt{q} \in \mathtt{Q}$. Note that this cannot be checked in general.

Now, if we have an extensional filter between two algorithmically equivalent languages, the following diagram commutes.



## 3.2 Breaking the equivalence

▶ **Theorem 2** (Extensional filtering discrimination). *Let* $\mathtt{L}$ *and* $\mathtt{M}$ *be two programming languages computing the same set of functions, that is* $[\![\mathtt{L}]\!] = \mathcal{F} = [\![\mathtt{M}]\!]$*. If there exists an extensional filter* $\mathtt{Q}$ *such that* $[\![\mathtt{L} \bigcap \mathtt{Q}]\!] = \mathcal{G} \supsetneq \mathcal{G}' = [\![\mathtt{M} \bigcap \mathtt{Q}]\!]$ *then* $\mathtt{L}$ *and* $\mathtt{M}$ *are not algorithmically equivalent.*

The fact that a filter is indeed extensional is seen as some kind of agreement between scientists as to what "implementing the same algorithm" should mean. Namely, the previous results are reread as:

- If we agree that programs with and without `cons` implement different algorithms (*i.e.* `\cons` is extensional), then there are more **algorithms** at high order than at first order.
- If we agree that programs with and without QI implements different algorithms, then there are more non-deterministic **algorithms** than deteministic one.

The strength of the result is that it allows to compare expressive power in the number of algorithms that can be implemented in a given language without giving a precise definition of algorithm. Whatever the notion of algorithm one wants to consider, under some reasonable assumptions, high-order or non-determinism allow to implement more algorithms.

Obviously, the assumptions about algorithms are subject to discussion. The filtering method allows the discussion to move from a still informal question (*"What is an algorithm?"*) to a more concrete discussion (*"Do algorithms need to respect QI?"*)

## References


1 Andreas Blass, Nachum Dershowitz, and Yuri Gurevich. When are two algorithms the same? *The Bulletin of Symbolic Logic*, 15(2), 2009.

2 Guillaume Bonfante. Observation of implicit complexity by non confluence. In *International Workshop on Developments in Implicit Computational complExity - DICE 2010*, Paphos, Cyprus, March 2010.

3 Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001.

4 Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776–2796, 2011.

5 Loic Colson. Functions versus Algorithms. *EATCS Bulletin*, 65, 1998. The logic in computer science column.

6 Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.

7 Serge Grigorieff and Pierre Valarcher. Classes of algorithms: Formalization and comparison. *Bulletin of the EATCS*, 107:95–127, 2012.

8 Neil D. Jones. *Computability and Complexity, from a Programming Perspective*. MIT press, 1997.

9 Neil D. Jones. The Expressive Power of Higher-order Types or, Life without CONS. *Journal of Functional Programming*, 11(1):55–94, 2001.

10 Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. *Fundamenta Informaticae*, 19(1,2):167,184, September 1993.

11 Daniel Leivant and Jean-Yves Marion. Predicative functional recurrence and poly-space. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97, Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 369–380. Springer, April 1997.