Search for Program Structure[*]

Gabriel Scherer

PRL Group, Northeastern University, Boston, MA, USA
`gabriel.scherer@gmail.com`

Abstract

The community of programming language research loves the Curry-Howard correspondence between proofs and programs. Cut-elimination as computation, theorems for free, `call/cc` as excluded middle, dependently typed languages as proof assistants, etc.

Yet we have, for all these years, missed an obvious observation: "the structure of *programs* corresponds to the structure of proof *search*". For pure programs and intuitionistic logic, more is known about the latter than the former. We think we know what programs are, but logicians know *better*!

To motivate the study of proof search for program structure, we retrace recent research on applying *focusing* to study the canonical structure of simply-typed λ-terms. We then motivate the open problem of extending canonical forms to support richer type systems, such as polymorphism, by discussing a few enticing applications of more canonical program representations.

1998 ACM Subject Classification F.3.3.4 Type Structure, F.4.1.2 Lambda Calculus and Related Systems, F.4.1.8 Proof Theory

Keywords and phrases programs, proofs, focusing, canonicity

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.15

1 Introduction

1.1 Canonical representations for arithmetic expressions in one variable

To introduce the idea of *canonical representation* of programs, let us start with an example in a simpler domain, simple arithmetic expressions over one fixed variable x.

$$\text{expression} \qquad a, b \quad ::= \quad n \in \mathbb{N} \mid a + b \mid a - b \mid a \times b \mid x$$

Suppose we are interested in the *meaning* of expressions, where the meaning of a is taken to be the evaluation function that takes in input the value of the variable x and returns the value of a with this parameter choice. You may think of inputs and outputs as natural numbers – or elements of a given ring.

Our presentation of expressions above has some nice properties, in particular it makes it obvious that the set of expressions is closed over addition and multiplication; it is easy to compose expressions from smaller expressions. But it also admits pairs of expressions that are syntactically distinct but have the same meaning, such as $2 + 2$ and 4, or $(a + b) \times c$ and $a \times c + b \times c$. We call these pairs *redundancies*.

There exists another common representation of these expressions: by applying simplification rules that preserve meaning, we can turn such expression a into a *polynomial* such as, for example, $2x^3 + 4x - 1$, which can be described formally as either 0 or a non-empty sum

[*] This research was supported in part by the National Science Foundation (grant CCF-1422133).

$\sum_{0 \leqslant k \leqslant d} c_k x^k$ of *monomials* of the form $c_k x^k$ with $c_k \in \mathbb{Z}$ and $c_d \neq 0$, for a natural number d called the *degree* of the polynomial.

We see our arithmetic expressions and the polynomials as two different *representations* of the same sort of objects – their meanings as evaluation functions. Polynomials, as a syntactic representation, have less redundancies than our expressions: $2+2$ and 4 are distinct expressions, but they are represented by the same polynomial $4x^0$. When a representation has less redundancies than another, we say that it is *more canonical.* In fact, one can easily show that polynomials have no redundancies at all: if two polynomials are syntactically distinct, they have distinct meanings. When a representation has this property, we say that it is *canonical.*

A more canonical representation has more structure: its definition encodes more of the meaning of the objects being represented. A canonical representation is very rigid, a lot of information about the object is apparent in its syntax, which often makes it easier to operate on it. For example, it is non-obvious whether an arithmetic expression a is constant (for all values of x), while this question can be easily decided for polynomials – it is constant when it is 0 or its degree is 0. In general, polynomials are much more convenient to manipulate for virtually any application, and they are the ubiquitous choice of representation when studying these objects.

1.2 Canonical representations of simply-typed λ-terms and its applications.

In this article, we discuss the design and use-cases of canonical representations of programs expressed in typed λ-calculi, which are more complex than arithmetical expression: finding a good notion of canonical representation can already be challenging.

Because canonical representations reveal so much, in their structure, of the meaning or *identity* of the programs, we should expect them to play a central role in answering many questions about programs. However, until recently theoretical difficulties hampered such practical applications.

In Section 2 (Theory), we present a recent brand of work that brought a much better understanding of canonical forms in the simply-typed case, inspired by ideas from *logic* and *proof theory*, in particular *focusing* and *saturation.* This inspiration is an instance of the Curry-Howard correspondence between proofs and programs that is different in nature from the its previous use-cases that are familiar to functional programmers, and closer to some formal approaches to logic programming. We then discuss why extending these representations outside the simply-typed systems – to type systems with polymorphism, closer to realistic programming languages or proof assistants – is difficult and interesting future work.

In Section 3 (Practice), we discuss potential applications of canonical representations of typed programs.

- Full abstraction, discussed in Section 3.1. Full abstraction, as a formal property of a translation from a source to a target language, requires that the translation preserves the equivalence between source programs. We discuss how having canonical representations on the source gives surprisingly strong full abstraction results; rather than a practical application, this gives new ways to think about full abstraction results and their consequences, by giving a new family of fully-abstract translations that behave in a fairly different ways from those obtained through more typical proof techniques.
- Equivalence checking, discussed in Section 3.2. Being able to mechanically check for equivalence opens the door to many interesting practical applications, such as automati-

cally verifying that a "refactoring" change indeed preserved the meaning of the program as intended – a somewhat tedious task that human reviewers currently have to perform – and solving some ambiguity situations or conflicts arising from a "diamond inheritance" situation. Because we know that for general programming language the question of equivalence fatally becomes undecidable, we must be careful to consider applications where "time out" is an acceptable answer.

- Program synthesis, studied in Section 3.3. Type-directed program synthesis algorithms try to enumerate all expressions of a certain type until they find one that satisfies certain user-provided conditions, such as input-output pairs or unit tests. More canonical representations eliminate redundancies – two syntactically distinct expressions of the same program behavior – so using them would seem very beneficial for program synthesis, by reducing the search space of program expressions to search. In fact, as we will show, existing work on program synthesis used some simplfications presented as heuristics, that are all instances of the simplifications leading to focusing-based more canonical representations.
  On the other hand, canonical representations may also require additional book-keeping, and to a certain point decrease raw efficiency of term enumeration. We propose studying these situations where de-normalization is desirable, starting from canonical representations, rather than studying partial heuristics starting from naive program representations.

2 Theory

2.1 Functional and logic programming

We have lived for too long in a world where logical justifications for programming language research were separated in two disconnected areas:

- *Functional programming*, whose computational behavior is given by term *reduction*, corresponding in logic to *cut-elimination* or in general elimination of detours in proofs.
- *Logic programming*, whose computational behavior is given by *proof search*.

It is natural for functional programmers to think about *complete* proofs that correspond to the program terms they are familiar with. They study relations between proofs or between programs, typically by a *reduction* relation that expresses computation, and an *equivalence* relation that expresses indistinguishability.

Curry-Howard then transfers intuition back and form between *program*-formers with a computational interpretation and *proof*-formers with an interpretation as a reasoning principle. Some success stories include relating control operators to classical reasoning principles [12], functional-reactive or event-driven programming to linear temporal logic [13, 23], session types to linear logic propositions [3], and building proof assistants with convenient computational principles out of advanced dependent type theories (Agda, Coq, LF, PRL...).

Logic programming considers *partial proofs* – derivations with some unfilled subgoals left to prove – to describe the state of a computation that evolves by proof search [18], either stopping when a complete proof is reached or enumeration all possible completions. *Proposition*-formers embed certain search principles that give rise to new computational behavior, possibly bound to the choice of well-designed search strategies. Besides proving a rich generalization of Prolog that is pleasing to proof theorists and type theorists alike, success stories include using linear logic to specify concurrent and distributed systems [16, 26] or to study interactive fiction gameplays and interactive storytelling [17].

We encourage functional programmers to start caring about proof search as well. When logicians propose a new strategy for proof search in a given logic, it avoids traversing redundant derivations of the same proof. In terms of programming, it removes redundant expressions of the same program. This lets functional programmers better understand the structure and identity of programs, by suggesting more canonical representations.

2.2 Identity in logic and programming

We make a careful distinction between a *program*, that is a specific behavior that the user has in mind when she implements it in a programming language, and its *expression* as a specific term in a given system of *representation* – defined by a syntax, typing rules, etc. A program can have arbitrarily many expressions; we say that two expressions are *equivalent* if they represent the same program. For example, refactoring is the process of moving from one expression of a program to another, that is judged more amenable to further modification.

In comparison to the arithmetic expressions of Section 1.1, programs (program behaviors) correspond to evaluation functions, and expressions correspond to terms, either in the syntax of simple arithmetic expressions – one choice of representation – or as polynomials – another choice.

▶ **Definition 1** (Canonicity). A representation T is *more canonical* than a representation S if they represent the same programs, and there is a mapping $\lfloor \_ \rfloor : T \to S$ such that the equivalence classes of T (the maximal sets of expressions of the same program) are mapped by $\lfloor \_ \rfloor$ to subsets of the equivalence classes of S. For example, β-normal λ-terms are more canonical than arbitrary λ-terms. A representation T is *canonical* if each equivalence class is a singleton – all equivalent representations are syntactically equal. For example, if you take a reasonable definition λ-calculus with integers, and restrict yourself to closed terms of type integer, then β-normal forms are canonical.

For programmers there is usually an extremely clear definition of the behavior of an expression, and therefore of what it means for two expressions to be equivalent. It may depend on what they decided to observe – they may, for example, ignore or pay attention to time and memory consumption – but is non-controversial as soon as the observable are agreed upon.

Logicians also make a distinctions between the expression of a formal proof and the mathematical proof it represents, and may consider that two derivations, two formal proofs are "morally the same". But how to define this distinction is not at all obvious: ordinary users of proofs do not consider their *behavior*, they are merely satisfied that they exist at all and could not care less about their formal identity.

How to explain, then, the following state of affairs? The identity of proofs has been studied for a long time by logicians, who proposed many different means of representing a mathematical proof, meant to better capture its identity, to strive at canonicity: natural deduction, sequent calculus, hypersequents / deep inference, focused sequents, tableaux, connection matrices, proof nets, etc. On the other hand, λ-terms reign as the ubiquitous representations of functional programs and, besides normalization of closed programs, the question of choosing alternative means of expression is rarely considered.

▶ Remark. We propose a tentative explanation – an excuse: canonicity is bad for programming. The choice of a highly non-canonical representation of programs allows programmers rich means of expressions of the same program. Two distinct expressions of the same program may correspond to stylistic difference, or an expression of intent in the way the program being worked on will evolve over time. Non-canonicity may be essential for flexibility, clarity and modularity. This is similar to the fact that human mathematicians do, in fact, use products $P \times Q$ of arbitrary polynomials P, Q when it is what they want to express, despite this not within the grammar of polynomials themselves – the product is only a simple arithmetic expression, which is simpler to *produce*. Yet, just as polynomials, if we want to write tools to *consume* programs, better understanding canonical representations can be essential.

2.3 Focusing for canonicity of simply-typed effectful λ-calculus

Focusing is based on the logical notion of *invertibility*. An inference rule is *invertible* when applying it during proof search cannot get you stuck: its premises are provable if and only if its conclusion is provable. Consider the logical rules corresponding to construction of functions, pairs and sums:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2} \qquad \frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 + A_2} i \in \{1,2\}$$

The rule for functions and pairs are invertible, but the rule for sum is not: using it means making a choice, and it is possible to get into a dead-end – if you try to prove A_1 but only A_2 was provable. On the side of destructors, the opposite is true: deciding to apply a function may get you in a dead-end if you don't know how to build an argument for it, while eliminating a sum merely considers two possible outcomes, losing no information.

Focusing, originally introduced for linear logic in Andreoli [1], is a series of conditions on proofs: some proofs are valid *focused proofs*, and other are not. First, if invertible rules can never get you in a dead-end, you can make them mandatory: focusing forces them to be applied eagerly, as much as possible. Second, once you can only apply non-invertible rules, the user has to select a formula (of the context or the goal), and focused imposes that apply non-invertible rules be applied to it as long as possible. For example, if the goal is $(X_1 + X_2) + (Y_1 + Y_2)$, focusing demands not only that they make a (non-invertible) choice for the head sum connective, but also choose between one of the X_i (or Y_i) by doing a nested sum introduction. As another example, if a function variable $f : X \to Y \to Z$ is in the context, selecting it means that one has to pass all arguments at once, passing both a X and a Y argument – stopping after applying X to work on another formula would be an invalid focused proof. Intuitively, using this function is only useful to get the final result Z, so it is always possible to either never apply it (if Z is not needed), or delay the application of X until Y is also available.

Zeilberger [31] proposed to consider focused proofs as a *syntax* for a programming language, guided by the observation that focusing could justify the dual semantic restrictions on polymorphism in effectful call-by-value and call-by-name languages – this works explains the value restriction for strict intersection types and the context restriction for lazy union types appearing in Dunfield and Pfenning [8], and the explanation was extended to second-order universal and existential polymorphism in Munch-Maccagnoni [19].

The idea of invertibility is one way to understand why adding sums make equivalence harder. We say that a type connective is *positive* if its right introduction rule is non-invertible, and *negative* otherwise: $(\to, \times, 1)$ are negatives, and $(+, 0)$ are positive. It is easy to decide equivalence of the simply-typed λ-calculus with only connectives of one polarity: we previously remarked that it is easy to define canonical forms in the negative fragment $\mathsf{STLC}(\to, \times, 1)$, but it is equally easy in the positive fragment $\mathsf{STLC}(+,0)$. It is only when both polarities are mixed that things become difficult.

A key result of Zeilberger [32, Separation Theorem, 4.3.14] is that a focusing-based presentation of the simply-typed λ-calculus is *canonical* in the *effectful* setting where we assume that function calls may perform side-effects – at least using the specific reduction strategy studied in CBPV [15]. Two syntactically distinct effectful focused program expressions are observationally distinct – the canonicity proof relies on two distinct error effects, to distinguish evaluation order, and an integer counter to detect repeated evaluation. The fact that any λ-term can be given a focused form comes from a *completeness* theorem, the analogous of completeness of focusing as a subset of logical proofs. However, this syntax is not canonical anymore if we consider the stronger equivalences of pure functional programming, where duplicated or discarded computations cannot be observed.

Let us write P, Q for positive types, types that start with a positive head connective, and N, M for negative types, that start with a negative head connective. In a λ-term in focused form with types of both polarities – see a complete description in Scherer [28, Chapter 10] – a non-invertible phases can be of two forms which we shall now define. It can start with a *positive neutral* $(p : P)$, which is a sequence of non-invertible constructors of the form $\sigma_i\ \_$ for positive types, they commit to a sequence of risky choice to build the value to return. Or it can start with a *negative neutral* $n[y : N]$, that performs a series of function applications $n\ p$ or pair projections $\pi_i\ n$ to a variable $y : N$ from the context. Such a negative neutral term, if it is of the type of an atomic goal $(n : X)$, is returned unchanged. If it is of some positive type P, it is bound to a variable name to be decomposed decomposed by the following invertible phase: $\texttt{let}\ (x : P) = n[y : N]\ \texttt{in}\ \ldots$. One way to think of these two choices is that positive neutrals $(p : P)$ build a part of the output value, while negative neutrals $n[y : N]$ observe a part of the input environment.

2.4 Saturation for local canonicity of simply-typed pure λ-calculus

Focusing enforces a form of *backward* search structure on canonical program representations: the invertible structure of terms is purely determined by the types of the goal and the context. To obtain canonical forms for pure terms, Scherer and Rémy [30] had to use intuitions from proof search again, by forcing the non-invertible part of terms to have the structure of a *forward* search: at the end of each invertible phase, try to *saturate* the context by making as much observations $\texttt{let}\ (x : P) = n[y : N]\ \texttt{in}\ \ldots$ as possible, returning a value $(x : P)$ only when introducing new observations does not teach us anything new. We call terms that follow this discipline *saturated terms* – for a more detailed explanation of saturation, see Scherer [28, Chapter 11].

There is no redundancy among saturated terms, but the completeness theorem is weaker than for focusing alone. To have the saturation process terminate, one has to fix in advance a finite set of possible observations for each context. For any such choice, only finitely many λ-terms can be given a saturated normal form – and conversely, for any finite set of λ-terms there exists a saturation strategy that allows to represent them as saturated normal form. This is enough for practical applications – testing whether a type has a unique inhabitant, testing whether two terms are equivalent (two is finitely many) – but still a technical limitation.

3 Practice

These results on the simply-typed λ-calculus are very encouraging, but they do not provide canonical forms in presence of *parametric polymorphism*, which gives even stronger equational principles. In other words, canonical forms for practically used programming languages are still out of reach.

We argue that building them would give us a much better understanding of what our programs *are*, and open the door for important applications.

1. Easy ways to build fully abstract translations.
2. Program synthesis.
3. Equivalence procedures.

3.1 Full abstraction

A translation from one program representation to another, a compiler for example, is said to be *fully abstract*[1] if two programs are observationally equivalent if and only if their translations are observationally equivalent.

Full abstraction is a strong property that gives a lot of information of a compilation process: it tells us that any equational reasoning at the source level remain valid at the target level. One can think of it as a usability property (knowing the source language is enough) or a security property (the compiler protects programs from being separated by attacks at the target level)

As a result of its strength, it is a difficult property to establish : many translations that we intuitively believe are fully-abstract have only been proved so very recently, or are not known to be. For example, fully-abstract translations from the simply-typed λ-calculus to the untyped λ-calculus or from the simply-typed λ-calculus (with recursive types) to System F were only established in 2016 [7, 21], and type-erasing fully-abstract translations from System F are an open problem. Yet we can, with apparent ease, provide you with very strong results.

▶ **Theorem 2** (Full abstraction for free!). *Take any finite subset S of the pure simply-typed λ-calculus with functions, pairs, sums and units. Take any system T in which any λ-term can be embedded: the untyped λ-calculus, System F, impure System F with references and `call/cc`, the Calculus of Constructions... There is a fully-abstract translation from S to T.*

Proof. For any pure λ-term e in S, let us define $\lfloor s \rfloor$ its canonical form as defined in Scherer [28], seen as a λ-term. We assumed that the λ-calculus embeds into T. We translate s into the embedding of $\lfloor s \rfloor$.

By definition of canonicity, if two terms s, t are observationally equivalent, then $\lfloor s \rfloor$ and $\lfloor t \rfloor$ are syntactically equal, and in particular their embedding in T is the *same* program. In other words, this translation is fully abstract. ◀

This result is a bit puzzling[2]. For example, we are not requiring the type $A \to B$ of pure functions in the pure simply-typed λ-calculus to be translated in T into a type of pure functions: the embedding into ML with non-termination and references, for example, would return a term at the type of ML impure functions. Yet the translation is fully-abstract.

Full-abstraction proofs are delicate because they are often built on the ability to *back-translate* contexts of the target language into the source language – completely or approximately. The proof technique established in Devriese, Patrignani, and Piessens [7] might be able to back-translate contexts of the calculus of (inductive) construction, but it sounds like a daunting amount of work. Instead, we paid the cost of coming up with canonical

[1] Full abstraction was originally introduced as a quality criterion for denotational semantics instead of syntactic program translation. We mention this in Section 4.2 (Game semantics).

[2] Note that it is not an immediate consequence of strong normalization for the simply-typed λ-calculus, otherwise it would extend to System F as well.

representations in the first place; once you have them for your source language, you get fully-abstract translations for free!

3.2 Equivalence procedures

Canonical representations also give procedures to decide equivalence for a programming language: to check whether two representations are equivalent, one can check whether their canonical forms are syntactically equal. Conversely, it is often – but not always – the case that equivalence procedures can be read back as a way to describe canonical forms for a language.

We know surprisingly little about deciding program equivalence. Equivalence in presence of non-empty sum types has remained an advanced, somewhat active research topic since it was proved decidable in Ghani [11], and equivalence in presence of the empty type (zero-ary sums) was decided only recently [29]. We know that observational equivalence for System F is undecidable, but while the (equally undecidable) problem of type inference has been actively studied since at least Pfenning [24], we know of no research on equivalence algorithms for polymorphic calculi.[3] Similarly, an impressive amount of effort has been invested in automatically checking that an implementation respects a specification (despite the undecidability barriers), but almost none went into automatic checking of implementation equivalence – as if this question jumped straight from automata to process calculi, without ever touching functional programs.

We hope that studying proof search and canonical representations in presence of polymorphism will stimulate work on semi-decidable equivalence algorithms for powerful type systems. The applications of equivalence procedures are everywhere, and we will mention some of them:

1. Robust equivalence checking opens the door to automatic verification of refactoring code transformations. Refactoring edits are typically designed to be easily reviewable by human programmers, so they should be within reach of an equivalence algorithm, even if it is very incomplete. Note that this very quickly involves the tricky equivalence of sums: moving a condition test earlier or later in a program is an instance of β-expansion on booleans (a sum type).
2. ML module systems have generative and applicative functors, the later being characterized by the fact that two independent applications of the same functor to the same parameter return compatible modules. This means that two independent libraries that made the same instantiation choice are compatible, instead of having to be modified to depend on a common functor applications. However, equality of module parameters, "static equivalence", is implemented in a very syntactic and restrictive way, essentially requiring that both independent libraries refer to a shared definition of the *parameter*. To liberate independent libraries, we should let each define their parameter, and check that they are *equivalent*.
3. The question of equivalence is also underlying the problem of *coherence* of implicit elaboration mechanisms such as type-classes and implicits. To remain predictable, language designs try to guarantee that each implicit elaboration problem has a unique solution across the user code – either by imposing somewhat ad-hoc priority/ordering restrictions or by imposing non-modular conditions, such as imposing all implicit instances

[3] One notable exception is the work of Bernardy, Jansson, and Classen [2] on polymorphic testing, which is secretly about program equivalence.

to be declared at the toplevel. If we had reliable equivalence checking, we could for example allow local declarations under the condition that local instances remain coherent with outer instances – checking equivalence for all pairs of elaboration chains for the same instance – or automatically determine which global instances need to be locally disabled to preserve coherence.

4. In general equivalence checking is the proper design tool to handle "diamond inheritance" situations. If a declaration inherits from two classes exporting the same method definition, existing languages either impose an arbitrary choice, or fail, or require a renaming; renaming is correct, but cumbersome in the common case where both definitions are equivalent – sometimes in non-trivial ways. This is common for example when modeling mathematical hierarchies: a finite monoid is both a monoid and a finite set, which both exports (non-conflicting) definitions of a carrier set.

3.3 Program synthesis

In Scherer and Rémy [30] we used canonical forms for the simply-typed λ-calculus to answer the following question: when it is the case that a given type is has a *unique* inhabitant modulo program equivalence? This unicity situation is a perfect opportunity for program synthesis, as the code that programmer would have in mind is completely determined by the type information flowing from the context. We presented some interesting examples (unique inhabitants at non-trivial types of `lens` operators), but of course unicity rarely happens in common programs today, only in highly polymorphic library functions, or code written with rich dependent types [27].

More practical-minded work on type-directed program synthesis, such as the recent works of Osera and Zdancewic [22], Frankle, Osera, Walker and Zdancewic [10], and Polikarpova, Kuraj and Solar-Lezama [25], proceeds by enumerating arbitrary many programs at a given type, looking for one that satisfies a user-provided specification – expressed as a set of input-output examples, or a refinement type. Even if the specifications can be woven inside the search procedure to reduce the search space, there is still a combinatorial explosion in the number of synthesized AST nodes that makes scaling to rich programs extremely challenging.

By reducing the redundancy among the programs of a given size, canonical forms provide order-of-magnitude[4] reduction in search space. In fact, many of the experimentally-motivated space reduction heuristics proposed in Osera and Zdancewic [22] are subsumed by focusing.

We propose three ways in which the study of canonical forms could interact and enrich future research on type-directed functional program synthesis:

1. We believe that instead of starting from a λ-term enumeration and carving out to reduce the search space, synthesis algorithms could be usefully expressed as directly from the search procedure for canonical forms – if they are known for the type system at hand – or the most-canonical representation known.
2. On the other hand, sometimes strong canonicity imposes some bookkeeping that can actually degrade search performance – Frankle, Osera, Walker and Zdancewic [10] reports that sometimes they willingly *avoid* η-expansion to reduce term size. Syntactic descriptions of "weakly" focused programs that do not impose full deep inversion has been proposed [20], but we need to build empirical and theoretical understanding of *when* and *why* de-canonicalization is useful for synthesis.

[4] The order-of-magnitude claim comes from Frankle, Osera, Walker and Zdancewic [10].

3. Finally, synthesis algorithms today mix techniques that are easily explained by the *proof search*-inspired point of view we promote in the present article, and others, such as the *predicate interpolation* used in Polikarpova, Kuraj and Solar-Lezama [25], that are not justified in these terms. It would be interesting to find justifications for them in purely logical terms – as logicians have historically have been able to do for various aspects of logic programming, such as forward vs. backward-search [5] or magic sets [4].

4 Advanced topics

4.1 Full abstraction seen differently

In Theorem 2 (Full abstraction for free!) we demonstrated that knowing a canonical representation for a source language lets us build fully-abstract translation to many different target languages with almost no assumption of those targets.

To be clear, we do not currently expect this approach to scale to the realistic languages we are interested to get fully-abstract compilers and translations for, because of the decidability barrier for canonical representations. Getting a *more canonical* representation is useful for the applications mentioned so far, but this approach to full abstraction requires fully canonical representations, and may therefore remain restricted to idealized languages. Nonetheless, in the rest of this section, we explore what we can learn about full abstraction from this extreme setting.

This full-abstraction argument is interesting because it gives a concrete example of the general idea that full-abstraction results from a source to a target language need not be built on back-translation techniques. Most of the known full-abstraction results rely on the ability to back-translate some parts of target terms and contexts – for example those whose type is the translation, in the target type system, of a source type – to the point where practitioners would sometimes conflate full-abstraction with the ability to back-translate. We demonstrate that knowing more about the source equivalence lets you work less on the relation between the source and target languages.

Full abstraction is also often seen as a security property, and considering our proof technique in this light is interesting. If we consider distinguishing two terms as an "attack", the statement of full abstraction says that the translation protects the source terms of any additional distinguishing power (attacks) in the target language; this may be achieved by having the compiler insert some sort of "protective wrapper" around the translated terms, to disable the more low-level (malicious?) features of the target language – see Fournet, Swamy, Chen, Dagand, Strub and Livshits [9] for example.

In our canonicity-based full abstraction result, there is no protective layer (neither static or dynamic) around the terms. Instead of disabling the observation features of the target language, we make sure that each source term will behave as all others equivalent terms under any additional observation from the target.

For example, consider the type $(X \to Y) \to X \to Y \to Y$ in the pure simply-typed λ-calculus and its two inhabitants $\lambda f\, x\, y.\, f\ x$ and $\lambda f\, x\, y.\, y$. To get a fully-abstract translation into a target calculus with side-effects such as input-output, one would typically wrap the two terms into a protective barrier that prevents using side-effects to tell if the f argument is used or not – in this example, the two terms are not equivalent, so they will remain distinguishable, but the protective barrier would be used in any case. This protection could be static, using a type of pure functions in the target language to forbid effects in f, or dynamic, using some kind of IO mocking/reification to block external observers from seeing writes. In our translation, no such protection is used; instead, saturation transforms both terms, in the source language (before translation), into terms of the form $\lambda f\, x\, y.\, \texttt{let}\ y' = f\ x\ \texttt{in}\ \ldots$, where the $\ldots$ are y' for the first program and y for the second. No protective wrapping is necessary anymore: bad f-observing side-effects will make the same observation on both normal forms.

Finally, one may wonder about the efficiency impact of the normalization into a canonical form – in particular the saturation. At the source type $(X \to Y) \to X \to Y \to Y$ in the example above, we turned $\lambda f\, x\, y.\, y$ into $\lambda f\, x\, y.\, \texttt{let}\ y' = f\ x\ \texttt{in}\ y$, introducing an apparently useless computation. In the general case, many such observations of the context may be introduced by saturation – how much depends on the finite set of terms that delimits the observable horizon. However, notice that this price needs only be paid if the function type $X \to Y$ is translated into a type of impure functions in the target. If it is translated to a pure function type, then the unused binding $\texttt{let}\ y' = f\ x$ can be removed after embedding in the target language. In general, saturation may introduce many additional observations, but only the ones that are required for full abstraction to hold need to be preserved; simple type-directed target simplification strategies may be able to convert back to target programs much closer to what non-canonicity-based fully-abstract compiler would produce.

4.2 Game semantics

Some readers may wonder about a relation between *canonicity* of syntactic representations that we propose to discover using *focusing*, and the *full abstraction* property of some denotational semantics obtained through *game semantics*. The comparison is as follows.

Game semantics provide a way to describe programs as *games* or, more precisely, *strategies* (as mathematical objects in a denotational semantics, but they could be given a concrete syntax), that allow to express an extremely broad scope of computational behaviors. In particular, two distinct such objects are distinct behaviors (there is no redundancy), but in general a lot of those objects do *not* correspond to interpretations any program of the language you are starting with: they may exhibit various kind of non-determinism, state, demand sensitivity, all sorts of weird things. Then you progressively carve out subsets of those objects that have nice behavioral properties (for example, innocent strategies), ruling out the strangest behaviors. If you do this enough, you can manage to remove all the non-standard stuff, and the remaining objects are exactly the programs of your language: you have a fully-abstract model.

The path to canonical forms goes in the opposite direction. We start with a syntax that, obviously, contains only expressions of the programs of our language, but a lot of these expressions may describe the same behavior. Then we carve out redundant expressions by proposing more and more canonical representations. If we do this enough, we can manage to remove all the redundancy, and the remaining expressions are exactly the programs of our language: we have a canonical representation.

In the best case, a language is so well-understood that both processes have reached completion: we have canonical representations and fully abstract denotations of it that are in one-to-one correspondence. This has been worked out for propositional linear logic in Laurent [14].

5 Conclusion

Syntax is a powerful tool to study and manipulate program behavior, and we emphasize the problem of finding *more canonical* syntactic representations of our program, that have less redundancy and are thus closer to a canonical description of their behavior.

We claim that this research programme can take heavy inspiration from proof-theoretic results and methods to study the structure of *proof search*, extending a new arm of the Curry-Howard correspondence.

Recent results have already been obtained for simply-typed systems, giving precise descriptions of normal forms for both effectful and pure λ-calculi. The extension to parametric polymorphism, and other advanced type systems (dependent types...) is an open problem that you should consider working on!

Powerful type systems, with their stronger equational reasoning principles, should not only make programming more expressive and safer, they should make it *easier*. We need better programming tools for that to happen, and hopefully more canonical representations are one way to make this happen.

Acknowledgments. Amal Ahmed led an extremely stimulating discussion on the nature and limitation of the full abstraction results provided by factorization through canonical forms in Theorem 2, which we tried to summarize in Section 4.1 (Full abstraction seen differently).

References

1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3), 1992.

2 Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In *ESOP*, 2010.

3 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, 2010.

4 Kaustuv Chaudhuri. Magically constraining the inverse method using dynamic polarity assignment. In *LPAR*, 2010. URL: `https://hal.inria.fr/inria-T00535948`.

5 Kaustuv Chaudhuri, Frank Pfenning, and Greg Price. A logical characterization of forward and backward chaining in the inverse method. *J. Autom. Reasoning*, 40(2-3), 2008.

6 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP*, 2000.

7 Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully abstract compilation by approximate back-translation. In *POPL*, 2016.

8 Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *POPL*, January 2004.

9 Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Évariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *POPL*, 2013. URL: `http://research.microsoft.com/pubs/176601/js-Tstar.pdf`.

10 Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, 2016.

11 Neil Ghani. Beta-Eta Equality for Coproducts. In *TLCA*, 1995.

12 Timothy G. Griffin. A formulae-as-type notion of control. In *POPL*, 1989.

13 Alan Jeffrey. Ltl types frp: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *PLPV*, 2012.

14 Olivier Laurent. Syntax vs. semantics: a polarized approach. *Theoretical Computer Science*, 343(1–2):177–206, 2005.

15 Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *TLCA*, 1999.

16 Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *PPDP*, 2005.

17 Chris Martens. Ceptre: A language for modeling generative interactive systems. In *AIIDE*, 2015.

18 Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 1991.

19 Guillaume Munch-Maccagnoni. Focalisation and Classical Realisability. In *CSL*, 2009.

20 Guillaume Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Université Paris Diderot, 2013.

21 Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *ICFP*, 2016.

22 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

23 Jennifer Paykin, Neelakantan R. Krishnaswami, and Steve Zdancewic. The essence of event-driven programming. In *draft*, 2016.

24 Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP*, 1988.

25 Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, 2016.

26 Anders Schack-Nielsen and Carsten Schürmann. Celf – A logical framework for deductive and concurrent systems (system description). In *IJCAR*, 2008.

27 Gabriel Scherer. Mining opportunities for unique inhabitants in dependent programs. In *DTP Workshop*, 2013.

28 Gabriel Scherer. *Which types have a unique inhabitant?* PhD thesis, Université Paris-Diderot, 2016.

29 Gabriel Scherer. Deciding equivalence with sums and the empty type. In *POPL*, 2017.

30 Gabriel Scherer and Didier Rémy. Which simple types have a unique inhabitant? In *ICFP*, 2015.

31 Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 2008.

32 Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.

A Appendix: Noam's history of focusing

Noam Zeilberger's work contains, expressed very clearly, the result that focusing gives a canonical term language for effectful programs – Zeilberger [32, Separation Theorem, 4.3.14]. Interestingly, this aspect is not emphasized in the general work, not is the connection to logic programming (despite the direct connection to Frank Pfenning's work), and the formulation in terms of a Curry-Howard correspondence between term canonicity and proof search is not to be found in his thesis – it seems to be a contribution of the present article.

We tried to guess how Noam Zeilberger ended up working with focusing and its applications to functional programming problems. We assumed that familiarity with focusing came first from the literature on logic programming, and applications to the simply-typed λ-calculus came only later. We asked him, and it turns out we were completely wrong. Most of his reply, lightly edited, is quoted below:

> The original reason I started thinking about evaluation order and pattern-matching was actually to understand intersection and union types. I was intrigued by the paper on "Tridirectional typechecking" by Joshua Dunfield and Frank [8], where they showed that the union-elimination rule requires an "evaluation context restriction" in the presence of effects, dual to the value restriction on interesection-introduction which Rowan Davies and Frank had found was necessary for CBV+effects ("Intersection types and computational effects", [6]). I was not

completely satisfied with the treatment of subtyping in these papers, though. Rowan and Frank's paper explained that the subtyping distributivity law

$$(A \to B) \wedge (A \to C) \leqslant A \to (B \wedge C) \tag{1}$$

from classical intersection type systems is unsound for CBV+effects, while Joshua and Frank's paper showed that the dual rule for unions

$$(A \to C) \wedge (B \to C) \leqslant (A \vee B) \to C \tag{2}$$

is unsound for CBN+effects (they didn't mention this explicitly, but it can be deduced from their examples showing the necessity of an evaluation context restriction on union-elimination). The solution in both papers was to simply drop these principles (1) and (2), but it's a fact that (1) is *sound* for CBN even in the presence of effects, while (2) is sound for CBV+effects. What I noticed (which really is kind of folklore) was that it's possible to reduce these questions of subtyping to typing by taking an "identity coercion" interpretation of subtyping. For example, it is possible to "derive" principle (1) by typing the eta-expansion of a function variable:

$$f : (A \to B) \wedge (A \to C) \vdash \texttt{fn x => f(x)} : A \to (B \wedge C)$$

The point is that the corresponding typing derivation goes away if we place a value restriction on intersection-introduction, because the subexpression `f(x)` is not a value. My hypothesis was that it should be derive *all* of the subtyping laws which are sound for a given evaluation order by typing the appropriate identity coercion, but this places constraints on the design of the type system, that the typing rules be "sufficiently complete". In particular, subtyping rules such as

$$\begin{array}{ll} (A * B) \wedge (A * C) \leqslant A * (B \wedge C) & \quad (A + B) \wedge (A + C) \leqslant A + (B \wedge C) \\ A * (B \vee C) \leqslant (A * B) \vee (A * C) & \quad A + (B \vee C) \leqslant (A + B) \vee (A + C) \end{array}$$

seemed to require inversion principles based on pattern-matching to be built into the typing rules. It took a few more steps to get to focusing. Here's a followup mail I wrote to Robert Harper on June 1 2006:

> I think I understand what you were getting at earlier – would you find it more acceptable to speak of call-by-value vs call-by-name *connectives*? Or maybe strict vs lazy connectives? Part of what this work is about is that we have to treat (for example) strict product and lazy product as different logical connectives, with different introduction and elimination rules. You can pattern-match against values of "strict type", and pattern-match against *covalues* of "lazy type". If you are familiar with the idea of focusing, strict (aka "positive") connectives are synchronous on the right and asynchronous on the left, whereas lazy (aka "negative") connectives are synchronous on the left and asynchronous on the right.
>
> [...] It may not be necessary to segregate distinct "call-by-value" and "call-by-name" type systems as I did in the version of the paper you have, and instead just take their union as a system mixing strict and lazy connectives – I haven't worked that out.