Proving the Equivalence of Higher-Order Terms by Means of Supercompilation*

Ilya Klyuchnikov and Sergei Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

Abstract. One of the applications of supercompilation is proving properties of programs. We focus in this paper on a specific task: proving term equivalence for a higher-order lazy functional language. The "classical" way to prove equivalence of two terms `t1` and `t2` is to write an equality function `equals` and to simplify the term `(equals t1 t2)`. However, this works only when certain conditions are met. The paper presents another approach to proving term equivalence by means of supercompilation. In this approach we supercompile both terms and compare supercompiled terms syntactically. Some applications of the technique are discussed. In particular, one of these applications may lead to the development of a more powerful "higher-level" supercompiler.

1 Introduction

The functional style of programming allows developers to write modular, maintainable and elegant programs. However, these advantages do not come for free. Making use of intermediate data structure, higher-order functions, lazy evaluation and function composition may result in a significant overhead during program execution. There are a number of program transformation techniques capable of eliminating this overhead. One of them is *supercompilation*, a technique suggested by V.F. Turchin in early 1970s. Initially, supercompilation was developed as a means of optimizing programs written in a functional language Refal [18,19], but later it was reformulated in more abstract terms [5,9,14,17].

Supercompilation is based on the following procedures:

- The construction of a labeled "process tree" that represents all possible traces of a computation process, the label (= "configuration") being a representation of the possible states of the computation.
- Decomposing and/or generalizing the configurations in order to turn the (possibly) infinite process tree into a finite graph.
- Generating the target ("residual") program from the graph.

Surprisingly, supercompilation turned out to be applicable not only to program optimization but also to program analysis and verification.

* Supported by Russian Foundation for Basic Research projects No. 08-07-00280-a and No. 09-01-00834-a.

A. Pnueli, I. Virbitskaite, and A. Voronkov (Eds.): PSI 2009, LNCS 5947, pp. 193–205, 2010.

Namely, transforming a program by means of a supercompiler may produce an equivalent target program, whose structure is, in a sense, "simpler" than the structure of the source program, so that some subtle properties of the source program may become readily apparent and easy to prove.

Moreover, if some knowledge may be formally expressed in terms of a program, supercompilation may be used for analyzing this knowledge and inferencing and making explicit some non-trivial, hidden facts.

Hence, supercompilation may play a role in program analysis similar to that of X-rays in radiography (at least, potentially).

It should be noted that there is a certain contradiction between the goals of program optimization and program analysis. The main goal of program optimization is to produce a program that is small and fast, but which may be absolutely unreadable for humans, being obscure, messy and ill-structured. Moreover, a program produced by an optimizer does not have to be strictly equivalent to the source one! If the source program successfully terminates for some input data and produces a meaningful result, the optimized program is certainly required to terminate and produce the same result. However, if the source program does not terminate, or terminates abnormally, the optimized program is often allowed to terminate or produce an arbitrary result (especially if this enables the optimizer to produce a faster and/or smaller target program). For example, the supercompiler SCP4 [11,12] often transforms functions extending their domains.

On the contrary, if a program transformation technique is used for program analysis, rather than program optimization, the programs produced by a transformer are not supposed to be executed. Thus, the size and execution speed of a transformed program is not a matter of great importance any more. In particular, the transformer does not have to try hard to avoid code duplication. For example, the following code

let p = f x y **in** g p q p r

can be safely transformed into this:

g (f x y) q (f x y) r

On the other hand, the preservation of program semantics may be highly desirable in cases where program transformation is used for the purposes of program analysis.

The present paper considers the problem of proving term equivalence by means of supercompilation. It is shown that some interesting classes of equivalencies can be proved by supercompiling both terms and comparing the supercompiled terms syntactically. It should be noted that this technique is applicable to languages with infinite data structures and higher-order functions. In addition, this approach does not require that a universal built-in equality predicate be present in the language.

Some applications of the technique are discussed. In particular, one of these applications may lead to the development of a more powerful supercompiler.

2 Why a Lazy Language with Higher-Order Functions?

Suppose there is some knowledge that is going to be encoded as a program, in order to be analyzed by a supercompiler. What programming language should be considered as "good" for this purpose? It could be argued that

1. The semantics of the language should be clearly defined.
2. The language should be easy for a supercompiler to deal with, especially if the supercompiler is required to strictly preserve the semantics of programs.
3. The language should be convenient for encoding knowledge as programs. In particular, infinite data structures are handy for representing infinite sequences of events and similar purposes.
4. The language should provide functions as first-class values. This is useful for formulating and proving "higher-order" assertions quantified over functions.

Since we are interested in reasoning about programs, and this is hardly possible for a language with obscure semantics, the first requirement is quite natural. Thus a functional programming language seems to be a good choice for our purposes.

The second requirement is easier to meet in the case of a lazy functional language, rather than a strict one, because many program transformation techniques (including supercompilation and deforestation) are "call-by-name" in nature. If these techniques are applied to a call-by-value language, the termination properties of programs may be violated. This, certainly, can be avoided by imposing some restrictions on input programs. For example, the termination properties are preserved by supercompilation, if the source program always terminates (see "total functional programming" [21]). Another approach is to impose certain restrictions on the transformations performed during supercompilation, which requires some additional analysis to be perform [9]. However, for the purposes of program analysis, the most straightforward solution is to just assume that the input language is a lazy one. In addition, for a lazy language the third requirement is met in a natural way.

The fourth requirement is motivated by the fact that in almost all programming languages a function's arguments are considered to be universally quantified. So a function definition can be read: for any x, y, ... If we deal with a first-order language then we can abstract over first-order data. But if we deal with a higher-order language, we can abstract over functions, too! Functions may represent rules, transformations, strategies and so forth.

In addition, there are cases where the results of supercompilation are just difficult to represent by a first-order program [16].

For the above reason, we have preferred to deal with a lazy functional language with higher-order functions.

3 HOSC: An Experimental "Higher-Order" Supercompiler

All experiments in program transformation described in the paper have been carried out by means of an experimental open-source supercompiler HOSC, dealing

$typeDef ::= typeCon = dataCon_1 \mid ... \mid dataCon_n$	type definition
$typeCon ::= tn\ type_1\ ...\ type_n$	type constructor
$dataCon ::= c\ type_1\ ...\ type_n$	data constructor
$type ::= tv \mid typeCon \mid type \rightarrow type \mid (type)$	type expression
$prog ::= typeDef_1 ... typeDef_n\ e$ **where** $f_1 = e_1 ... f_n = e_n$	program
$e ::= v$	variable
$\mid c\ e_1 ... e_n$	constructor
$\mid f$	function
$\mid \lambda v.e$	abstraction
$\mid e_0\ e_1$	application
$\mid$ **case** e_0 **of** $p_1 \rightarrow e_1 ... p_k \rightarrow e_k$	case term
$\mid$ **letrec** $f = \lambda v.e$ **in** e	local function
$\mid (e)$	term in parenthesis
$p ::= c\ v_1 ... v_n$	pattern

where tn ranges over type names, tv ranges over type variables, c ranges over constructors.

Fig. 1. HLL grammar

with a lazy language with higher-order functions[1]. HOSC *preserves* the semantics of programs, which is *essential* for the techniques described in the paper.

HOSC transforms programs written in HLL, a simple higher-order lazy language, similar to that used by Hamilton [6,7]. HLL is typed using the Hindley-Milner polymorphic typing system.

A program in HLL consists of a number of data type definitions, a term to be evaluated and a set of function definitions (see Fig. 1).

A left-hand side of a data type definition is a type name (more precisely, a type constructor name) followed by a list of type variables. The right-hand side consists of one or more constructor declarations.

The grammar of HLL is shown in Fig. 1. A term is either a variable, a constructor, a lambda abstraction, an application, a case term, a local function definition or a term in parenthesis. A function definition binds a variable to a lambda abstraction. The intended operational semantics of HLL is the normal-order graph reduction to a weak head normal form. The data analysis is performed by pattern matching with constructors in case terms (as in [17]).

A term in HLL may contain free variables and local function definitions.

Note that the construct `where` is only a syntactic sugar, since the function definitions can always be transformed to `letrec`-s and moved to the term preceding `where`. Hence, any program is essentially a single term (plus a number of type declarations), so that there is no difference between transforming a term and transforming a program. In particular, the equivalence of programs is just the equivalence of terms.

[1] See the HOSC web-application and the sources at `http://hosc.appspot.com`

4 Proving Term Equivalence

4.1 Proving Properties of Terms by Supercompilation

As shown by Turchin [19,20], some properties of programs can be proved by program transformation. For example, suppose there is a function f (represented as a program), and we want to prove that, for any input x, the result returned by f satisfies some property p. Then we may encode p as a program, and try to "simplify" the term $p(f(x))$ by means of a supercompiler. If the structure of the supercompiled term is trivial, so that it can be readily seen that the evaluation of the term never returns `False` and always terminates, we can conclude that the source term $p(f(x))$ always returns `True`. Therefore, the result of evaluating $f(x)$ always satisfies the property p.

The fruitfulness of this approach has been recently shown by Lisitsa and Nemythykh [12], who have succeeded in verifying a number of cache coherence protocols by means of the supercompiler SCP4.

4.2 Equality-Based Approach to Proving Term Equivalence

As pointed out by Turchin [18], proving the equivalence of two terms `t1` and `t2` can be reduced to proving a property of a single term. Namely, if `equals` is a function testing values for equality, we can compose the term `equals t1 t2` and supercompile it to see whether it always returns `True`.

Consider the program in Fig. 2 in which the function `plus` takes two natural numbers (in unary system) and returns their sum. We want to prove that

```
equals (plus (S x) y) (plus x (S y))
```

or, in more "mathematical" notation, that

$$(x + 1) + y = x + (y + 1)$$

The result of supercompiling the program is shown in Fig. 3. It can be readily seen that the supercompiled program never returns `False`. However, there remain a few subtle points concerning such kind of reasoning!

4.3 Restrictions and Drawbacks of the Equality-Based Approach

Suppose the term `equals t1 t2` never returns `False`. Does it mean that `t1` and `t2` are really "equivalent"?

It depends on what is understood by "equivalence". The "equality-based" approach to proving term equivalence is based on a number of assumptions:

1. There exists a built-in equality function `equals`, or, at least, `equals` can be defined for the values returned by `t1` and `t2`.
2. All data structures involved are finite.
3. The evaluation of `t1` and `t2` always terminates.

Assumption 1 is usually true of first-order strict languages (like Refal [18,12]). However, in the case of a higher-order language there arise some problems, because `t1` and `t2` may return functional values, which are impossible to test for equality.

```
data number = Z | S number
data boolean = True | False

equals (plus (S x) y) (plus x (S y)) where

plus = λx.λy.
    case x of
        Z → y
        S x1 → S (plus x1 y)

equals = λx.λy.
    case x of
        Z →
            case y of
                Z → True
                S y1 → False
        S x1 →
            case y of
                Z → False
                S y1 → equals x1 y1
```

Fig. 2. Proving $(x + 1) + y = x + (y + 1)$: the source program

```
data number = Z | S number
data boolean = True | False

letrec f = λp2.λr2.
    case p2 of
        Z → letrec g = λs2.
                    case s2 of
                        Z → True
                        S w → g w
            in g r2
        S p1 → f p1 r2
in f x y
```

Fig. 3. Proving $(x + 1) + y = x + (y + 1)$: the supercompiled program

Assumption 2 is not automatically true in the case of a lazy language (even a first-order one).

Assumption 3 may not be true in many interesting cases. For example, if `t1` and `t2` deal with infinite data structures and, by necessity, never terminate, but are still "equivalent" (i.e. have the same "meaning" according to the language's semantics).

4.4 Normalization-Based Approach to Proving Term Equivalence

In order to get rid of dealing with equality predicates, we need an alternative, more general, definition of term equivalence. Thus, the "contextual equivalence", as defined by Pitt [15], seems to be a reasonable choice:

> Loosely speaking, two expressions M and M' of a programming language are contextually equivalent if any occurrences of M and M' in complete programs can be interchanged without affecting the results of executing the programs.

In particular, the above definition implies that two programs are trivially equivalent, if they are "syntactically isomorphic" (i.e. identical, modulo some trivial renaming and/or rearranging of the constructs appearing in the program).

Let $A \Rightarrow_{sc} A'$ mean that A' is semantically equivalent to A and can be produced by supercompiling A, or, in other words, $\Rightarrow_{sc}$ is a "supercompilation relation" (as defined by Klimov [10]).

Let $\approx$ denote equivalence and $\cong$ "syntactic isomorphism" of programs. Then the following holds:

$$\frac{A \Rightarrow_{sc} A' \qquad B \Rightarrow_{sc} B' \qquad A' \cong B'}{A \approx B}$$

Or, in plain words, if supercompiling A and B results in producing essentially the same residual program, then A and B are equivalent.

Thus, supercompilation can be seen as transformation that, in a sense, "normalizes" terms. Some other program transformation techniques can also be considered as normalizing ones [1,2,3,4].

Note that the general idea of proving equivalence by normalization is a well-known one, being a standard technique in such fields as computer algebra. The idea of using supercompilation for normalization is due to Lisitsa and Webster [13], who have successfully applied supercompilation for proving the equivalence of programs written in a first-order functional language, provided that the programs deal with finite input data and are guaranteed to terminate.

> We argue that this techinque is also applicable to higher-order functional programs, even if they deal with inifinite data structures and do not terminate for some inputs.

Let us consider the program in Fig. 4 containing definitions of a few well-known functions over lists. Supercompiling the term `map (compose f g) xs` produces the program shown in Fig. 5. On the other hand, supercompiling the term `(compose (map f) (map g)) xs` results in the same residual program (modulo alpha renaming)! Hence, we have proved that the following holds

```
map (compose f g) xs = (compose (map f)(map g)) xs
```

for all `f`, `g`, and `xs` that are allowed by the type system of the language HLL. Note that this statement holds for all lists `xs` including infinite lists and $\bot$, whose elements may be quite exotic: first-order values, functions, data trees, or $\bot$. Also note that the functions `f` and `g` do not have to terminate.

> Therefore, the normalization-based approach enables us to prove statements that are even impossible to formulate in terms of the equality-based approach!

```
data List a = Nil | Cons a (List a)
data Boolean = True | False
data Pair a b = P a b

compose = λf.λg.λx.f (g x)
unit = λx.Cons x Nil
rep = λxs. append xs
abs = λf. f Nil
iterate = λf.λx. Cons x (iterate f (f x))
fp = λp1.λp2.
    case p1 of P a1 a2 →
        case p2 of P b1 b2 → P (a1 b1) (a2 b2)

map = λf.λxs.
    case xs of
        Nil → Nil
        Cons x1 xs1 → Cons (f x1) (map f xs1)

join = λxs.
    case xs of
        Nil → Nil
        Cons x1 xs1 → append x1 (join xs1)

append = λxs.λys.
    case xs of
        Nil → ys
        Cons x1 xs1 → Cons x1 (append xs1 ys)

idList = λxs.
    case xs of
        Nil → Nil
        Cons x1 xs1 → Cons x1 (idList xs1)

filter = λp.λxs.
    case xs of
        Nil → Nil
        Cons x xs1 →
            case p x of
                True → Cons x (filter p xs1)
                False → filter p xs1

zip = λp.case p of P xs ys →
    case xs of
        Nil → Nil
        Cons x1 xs1 →
                  case ys of
                    Nil → Nil
                    Cons y1 ys1 → Cons (P x1 y1) (zip (P xs1 ys1))
```

Fig. 4. Example functions over lists

```
data List a = Nil | Cons a (List a)
letrec
  h = λys.
    case ys of
      Nil → Nil
      Cons y1 ys1 → Cons (f (g y1)) (h ys1)
in
  h xs
```

Fig. 5. Supercompilation of *map* (*compose f g*) *xs*

The authors have implemented an equivalence checker based on term normalization and built on top of the specializer HOSC. Following are a number of sample equivalences that have been automatically proved by the checker:

```
compose (map f) unit = compose unit f
compose (map f) join = compose join (map (map f))
append (map f xs) (map f ys) = map f (append xs ys)
append (append xs ys) zs = append xs (append ys zs)
filter p (map f xs) = map f (filter (compose p f) xs)
iterate f (f x) = map f (iterate f x)
map (compose f g) xs = (compose (map f)(map g)) xs
rep (append xs ys) zs = (compose (rep xs) (rep ys)) zs
(compose abs rep) xs = idList xs
map (fp (P f g)) (zip (P x y)) = zip (fp (P (map f) (map g)) (P x y))
append r (Cons p ps) =
  case (append r (Cons p Nil)) of
    Nil → ps
    Cons v vs → Cons v (append vs ps)
```

Note that some of the above equivalences are instances of Wadler's "free theorems" [22,8].

Given the program in Fig. 2, the associativity of addition can be proved by supercompiling both sides of the equation

```
plus (plus x y) z = plus x (plus y z)
```

One might expect that the commutativity of addition

```
plus x y = plus y x
```

could be proved in the same way. However, this is not the case, just because the conjecture is not true! The language HLL is a lazy one, for which reason `plus (S Z)` $\bot$ `= (S` $\bot$`)`, but `plus` $\bot$ `(S Z)` $= \bot$.

5 Applications of the Technique

5.1 Generating Sets of Equivalent Terms

Since the set of all terms is recursively enumerable, it is possible to write a generator automatically producing sets of equivalent terms. A straightforward procedure may look as follows.

First, a potentially infinite sequence of term can be generated, the terms being ordered according to their size. Then the sequence of terms can be filtered, in order for the terms that are not well-typed to be rejected. Then the well-typed terms can be "normalized" by supercompiling them, and partitioned into equivalence classes by comparing their "normalized" versions.

Certainly, the above procedure is not "complete", because term equivalence is, in general, undecidable. Hence, for any given supercompiler, some equivalences will not be proved by supercompilation. However, an important point is that the above procedure is capable of automatically *discovering* equivalences, rather than just proving them.

5.2 Term Equivalence and Higher-Level Supercompilation

As has been shown above, given a supercompiler, a library of term equivalences can be generated. And this library can be used for increasing the power of supercompilation. In other words, we can build a "higher-level" supercompiler using a "classic" supercompiler as a "lower-level" building block.

Namely, if a "classic" supercompiler encounters two configurations A and B, such that A is homeomorphically embedded into B, the supercompiler tries to fold B to A. This is possible, if B is an instance of A. Otherwise, the supercompiler has to throw B away and replace A with a more general configuration, which may lead to "over-generalization".

However, given a library of equations, a "higher-level" supercompiler may replace B with an equivalent configuration B' that is an instance of A, so that B' can be folded to A.

As an example, let us consider supercompiling a naïve definition of the function `reverse` into one with an accumulating parameter (which is more efficient).

Let us try supercompiling the following term:

```
append (reverse xs) ys
```

After unfolding we get:

```
case reverse xs of
    Nil → ys
    Cons x3 x4 → Cons x3 (append x4 ys)
```
(1)

Further unfolding results in:

```
case
    case xs of
        Nil → Nil
        Cons x5 x6 → append (reverse x6) (Cons x5 Nil)
of
    Nil → ys
    Cons x3 x4 → Cons x3 (append x4 ys)
```
(2)

Now we have to split the configuration by considering two cases: `xs` = `Nil` and `xs` = `Cons x5 x6`. If `xs` = `Nil`, the configuration is reduced to `ys`, and, in the second case, it is transformed into

```
data List a = Nil | Cons (List a)

letrec reverse1 = λxs1.λys1.
    case xs1 of
        Nil → ys1
        Cons x2 xs2 → reverse1 xs2 (Cons x2 ys1)
in
  reverse1 xs ys
```

Fig. 6. Higher-level supercompilation of `append (reverse xs) ys`

```
case append (reverse x6) (Cons x5 Nil) of
    Nil → ys
    Cons x3 x4 → Cons x3 (append x4 ys)
```
(3)

The term (3) embeds the term (1), without being an instance of (1). Hence, a "classical" supercompiler would have to generalize (1). But the generalization can be avoided by using the following equation

```
append r (Cons p ps) =
case (append r (Cons p Nil)) of
    Nil → ps
    Cons v vs → Cons v (append vs ps)
```
(4)

Note that this equation can be proved by term normalization.

Applying the substitution {`r` = `reverse x6`, `p` = `x5`, `ps` = `ys`} to the above equality, we can replace the term (3) with the equivalent term

```
append (reverse x6) (Cons x5 ys)
```

which is an instance of the initial term `append (reverse xs) ys`. Hence, a folding can be performed, to produce the final result of this "higher-level" supercompilation shown in Fig. 6.

Therefore, the following equation has been proved

```
append (reverse xs) ys = reverse1 xs ys.
```

which implies that

```
reverse xs = append (reverse xs) Nil = reverse1 xs Nil.
```

The original definition of `reverse` was quadratic in the length of xs, while the transformed one is linear. Hence, the proposed technique is capable of producing results similar to those achieved by "distillation", another approach to "higher-level" supercompilation suggested by Hamilton [6].

6 Conclusions

We have shown that the equivalence of terms can be proved by means of supercompilation without the use of an equality predicate, which makes the technique

applicable to lazy languages with higher-order functions. The techniques can be used to increase the power of supercompilation, to achieve the results similar to distillation, which is another approach to "higher-level" supercompilation.

Acknowledgements

An early version of this work was presented as a talk at Copenhagen Programming Language Seminar at DIKU, and we would like to thank Robert Glück, Torben Mogensen and Andrzej Filinski for their useful comments and inspiring advices. The authors are also grateful to Geoff Hamilton, as well as Andrei Klimov and other participants of Refal Seminars at Keldysh Istitute of Applied Mathematics for fruitful discussions.

References

1. Albert, E., Vidal, G.: The narrowing-driven approach to functional logic program specialization. New Generation Computing 20(1), 3–26 (2002)
2. Alpuente, M., Falaschi, M., Vidal, G.: Partial evaluation of functional logic programs. ACM Transactions on Programming Languages and Systems (TOPLAS) 20(4), 768–844 (1998)
3. Cockett, R.: Deforestation, program transformation, and cut-elimination. Electronic Notes in Theoretical Computer Science 44(1), 88–127 (2001)
4. Dybjer, P., Filinski, A.: Normalization and partial evaluation. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 137–192. Springer, Heidelberg (2002)
5. Glück, R., Klimov, A.V.: Occam's razor in metacompuation: the notion of a perfect process tree. In: Cousot, P., Filé, G., Falaschi, M., Rauzy, A. (eds.) WSA 1993. LNCS, vol. 724, pp. 112–123. Springer, Heidelberg (1993)
6. Hamilton, G.W.: Distillation: extracting the essence of programs. In: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pp. 61–70. ACM Press, New York (2007)
7. Hamilton, G.W., Kabir, M.H.: Constructing programs from metasystem transition proofs. In: Proceedings of the First International Workshop on Metacomputation in Russia (2008)
8. Holst, C.K., Hughes, J.: Towards binding-time improvement for free. In: Functional Programming, Workshops in Computing, Glasgow. Springer, Heidelberg (1990)
9. Jonsson, P.A.: Positive supercompilation for a higher-order call-by-value language. Luleå University of Technology (2008)
10. Klimov, A.V.: A program specialization relation based on supercompilation and its properties. In: Proceedings of the First International Workshop on Metacomputation in Russia, pp. 54–78. Ailamazyan University of Pereslavl (2008)
11. Lisitsa, A., Nemytykh, A.P.: Reachability analysis in verification via supercompilation. International Journal of Foundations of Computer Science 19(4), 953–969 (2008)
12. Lisitsa, A.P., Nemytykh, A.P.: Verification as a parameterized testing (experiments with the scp4 supercompiler). Programming and Computer Software 33(1), 14–23 (2007)

13. Lisitsa, A.P., Webster, M.: Supercompilation for equivalence testing in metamorphic computer viruses detection. In: Proceedings of the First International Workshop on Metacomputation in Russia. Ailamazyan University of Pereslavl (2008)
14. Mitchell, N., Runciman, C.: A supercompiler for core haskell. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 147–164. Springer, Heidelberg (2008)
15. Pitts, A.M.: Operationally-based theories of program equivalence. In: Semantics and Logics of Computation, pp. 241–298 (1997)
16. Romanenko, S.A.: Higher-order functions as a substitute for partial evaluation. In: Proceedings of the First International Workshop on Metacomputation in Russia. Ailamazyan University of Pereslavl (2008)
17. Sørensen, M.H., Glück, R., Jones, N.D.: A positive supercompiler. Journal of Functional Programming 6(6), 811–838 (1993)
18. Turchin, V.F.: The Language Refal: The Theory of Compilation and Metasystem Analysis. Department of Computer Science, Courant Institute of Mathematical Sciences, New York University (1980)
19. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems (TOPLAS) 8(3), 292–325 (1986)
20. Turchin, V.F.: Metacomputation: Metasystem transitions plus supercompilation. In: Danvy, O., Thiemann, P., Glück, R. (eds.) Dagstuhl Seminar 1996. LNCS, vol. 1110, pp. 481–509. Springer, Heidelberg (1996)
21. Turner, D.A.: Total functional programming. Journal of Universal Computer Science 10(7), 751–768 (2004)
22. Wadler, P.: Theorems for free! In: FPCA 1989: Proceedings of the fourth international conference on Functional programming languages and computer architecture, pp. 347–359. ACM, New York (1989)