On the Place of Supercompilation inside Program Specialization

Andrei P. Nemytykh*

Program Systems Institute of Russian Academy of Sciences
nemytykh@math.botik.ru

Abstract. Research in the field of creating systematical methods for specialization of programs with respect to fixed properties of their arguments, compositional structure and given invariants were started by Russian scientists A. P. Ershov ("mixed computation"), V. F. Turchin ("supercompilation") and Japanese scientist Y. Futamura ("generalized partial computation") in the 1970-ths. To the current moment a huge amount of facts mainly related to the object domain of functional programming languages was accumulated in the literature.
Ideas of supercompilation were mainly being studied on the base of a functional programming language REFAL, although a series of the results were polished on the LISP's experimental base. At present time, along with a number of primitive supercompilers constructed for simplest purely theoretical languages, there exists the only experimental supercompiler SCP4 for a real programming language (REFAL-5). The name SCP4 was suggested by V. F. Turchin as reflecting the history of the supercompilation ideas.
In this paper we consider various approaches to formulation of the specialization task *per se*. We give a short survey of the main achievements derived (to the given moment) in the field of specialization of functional programs, analyze principal distinctions between supercompilation and other existing methods. We survey the attempts of constructing of supercompilers.

Keywords: Program transformation, program specialization, supercompilation, partial evaluation, REFAL.

1 Preliminaries

Definition 1. *An implementation of a functional programming language* $\Re$ *is a quadruple* $\langle P, D, U, T\rangle$*, where sets* $\mathtt{P}$, $\mathtt{D}$ *are called as a* $\Re$*-program set and a* $\Re$*-data set correspondingly; partial recursive functions* $U\colon\ P\times D \mapsto D$ *and* $T : P \times D \mapsto \mathbb{N}$ *are named correspondingly as a universal function (or semantics) and a time*

* The author is supported by Russian Foundation for Basic Research (grants 07-07-92100-GFEN_a, 08-07-00280-a), Program for Basic Research of Presidium of Russian Academy of Sciences (as a part of "Development of the basis of scientific distributed informational-computing environment on the base of GRID technologies") and Russian Federal Agency of Science and Innovation project No. 2007-4-1.4-18-02-064.

measure function of the $\Re$ language. Here $\mathbb{N}$ stands for the set of the natural numbers.

Below we use the shorthand notation `p(x)` for `U(p,x)`.

2 On Two Task Statements of Program Specialization

Two different statements of the specialization task *per se* are considered in the scientific literature. We will formulate them in natural precision terms. The difference between the concepts of a total recursive function and a partial recursive function is essential in the following statements of the tasks.

Let an implementation of a functional programming language $\Re = \langle \mathtt{P},\mathtt{D},\mathtt{U},\mathtt{T}\rangle$ be given, where $\mathtt{D} = \bigcup_{n\in\mathbb{N}} M^n$ for a nonempty set M.

The task 1. Let a program `p(x,y)` from `P` define a *partial recursive function* $F(x,y) : \mathtt{D}\times\mathtt{D} \mapsto \mathtt{D}$. Given a value of the first argument $x_0 \in \mathtt{D}$ of the function F, the specialization task requires to construct another program `q(y)` $\in$ `P` such that

$$\forall \mathtt{y} \in \mathtt{D}.(\mathtt{q(y)} = \mathtt{p}(\mathtt{x}_0,\mathtt{y})) \wedge (\mathtt{T(q,y)} \leq \mathtt{T(p,x}_0\mathtt{,y)}),$$

where the value `q(y)` determined if and only if the value `p(`$\mathtt{x}_0$`,y)` determined. Otherwise their non-determination types (abnormal stop or infinite evaluation time) must coincide. That is to say, in this task the programs `q(y)` and `p(`$\mathtt{x}_0$`,y)` define the same *parial recursive function*, namely $F(x_0, y)$.

The task 2. Let a program `p(x,y)` from `P` define a *total recursive function* $F(x,y) : X\times Y \mapsto \mathtt{D}$, where $X \subset \mathtt{D}, Y \subset \mathtt{D}$. Given a value of the first argument $x_0 \in \mathtt{D}$ of the function F, the specialization task requires to construct another program `q(y)` $\in$ `P` such that

$$\forall \mathtt{y} \in Y.(\mathtt{q(y)} = \mathtt{p}(\mathtt{x}_0,\mathtt{y})) \wedge (\mathtt{T(q,y)} \leq \mathtt{T(p,x}_0\mathtt{,y)}).$$

In the other words, in the second task the program `q` defines an extension of the *total recursive function* $F(x_0, y) : Y \mapsto \mathtt{D}$ onto the second argument.

The program `q(y)` is said to be a *residual* program.

The substantial part of the tasks is to construct an optimal `q`
(with respect to the running time).

Various specifications of the concept of *optimality* (the time measure function `T`) define concrete approximations of the specialization task *per se*. Roughly speaking, the first task demands that the residual program `q` has to preserve the operational semantics of the source program `p`. The second task is more natural from the point of view of applications: usually, operational behavior of the residual program does not matter for users, if the input data do not belong

to the users' subject domain. On the other hand, the conditions of the second task provide more freedom for concrete specialization methods. That frequently allows construct more optimal residual programs as compared with the methods restricted with the constraints imposed by the first task.

Supercompilation methods are oriented to solve the second task.

3 A Survey of the Results in the Field of Program Specialization

Great difficulties arose on the way of development and implementation of the basic ideas formulated by A. P. Ershov, V. F. Turchin and Y. Futamura. Later N. D. Jones (Denmark) suggested to weaken the originally set goals at the expense of the specialization methods [16,14]. This simplified technique known as partial evaluation is the most developed one to the given moment. It solves the first specialization task. Trying to solve the tasks of self-application of a specializer, also independently formulated by the three above mentioned researchers in the 70-ths, N. D. Jones together with his colleagues made still another principal step towards simplification of the specialization methods. In the 1985-th N. D. Jones, P. Sestoft and H. Søndergaard (University of Copenhagen) succeed in solving an approximation task of self-application of a Copenhagen partial evaluator `mix` [15]. Here we have to note that there exists always a time measure function `T` allowing to construct the following residual program:

$$\texttt{q(y) \{ = p(x}_0\texttt{,y); \}}$$

i.e. simply copying the source program `p` and fixing the given value of the first argument in the entry point of the `p`. The first results of self-application of `mix`, substantially, just slightly differed from the trivial residual program given above. In the 1995-th [11] N. D. Jones wrote that the length of the residual program obtained as a result of a simplest task of self-application

$$\texttt{mix(}\underline{\texttt{mix(p}_0\texttt{,x,y)}}\texttt{)}^1$$

of `mix` with respect to a given three-line program $\texttt{p}_0\texttt{(x,y)}$ was five hundred pages. Here values of `y` are unknown to both copies of `mix`; the value of the `x` argument is known to the `mix` being specialized, while it is unknown to the `mix` specializing the program $\texttt{p}_0$. Analyzing the residual program the Copenhagen group suggested the concepts of *"online"* and *"offline"* specialization methods [14]. Below we consider these concepts. The choice of the simpler "offline" methods allowed in the 1986-th to solve more reasonable tasks of self-application of the partial evaluator `mix` [14,35]. By means of introduction of tools rising the arities of the programs being specialized (as well as their subprograms) in the frames of the "offline" approach, in the 1987-th [31,33,34], S. A. Romanenko succeeded in substantial improvement of the structural and running time properties of the

[1] Here the underline denotes an encoding.

residual programs resulted in several tasks of self-application of the Moscow partial evaluator `unmix`. The following name of his paper describing `unmix` is self explanatory: "A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure" [31,33].

Offline specialization parts analyzing the source program `p` and metainterpretation of the local `p`'s steps (evaluation of which can be done without knowledge of the *concrete* values of the unknown part of the steps' arguments) in separate processing stages. The input for the first stage named as biding time analysis (BTA) is the `p` and information indicating the part of the `p`'s arguments, which will be known to the second stage of transformation (metainterpretation) rather than concrete values of the arguments, and the other part of the arguments, which will be unknown to the second stage. The first kind of the arguments is named as *static*, while the second kind is named as *dynamic*. The BTA's output is an annotated program $\mathtt{p}^{\mathtt{ann}}$, in which each elementary action is annotated as static whenever it can be unambiguously interpreted without knowledge of the concrete values of the dynamic part of the input of these actions-steps. The arguments of every such a step are annotated as static or dynamic as well. The BTA analyses the static information flow ("movement") along the program `p`. Obviously, the task (*per se*) formulated for the BTA is algorithmically undecidable. Everywhere here, by default, we mean some approximation of the task formulated for the BTA. The input for the second stage (which, in fact, is named just as *"specialization"* in such an approach) is the $\mathtt{p}^{\mathtt{ann}}$ together with the values of its static arguments. *"Specialization"* (the second stage) is logically simple and decidable; all substantial problems were moved to the BTA. The Jones' group, as well as S. A. Romanenko, solved the following task

$$\mathtt{mix}(\underline{\mathtt{mix}^{\mathtt{ann}}(\mathtt{p}_0^{\mathtt{ann}}}, \mathtt{x}, \underline{\mathtt{y}}))^2$$

rather than the original classical self-application task. The solved task is substantially simpler as compared with the classical one: both copies of `mix` perform only the second stage of transformation and do nothing concerning the biding time analysis. Later the Jones-Romanenko's offline self-application experiments were reproduced and made more accurate by a number of authors. Here is substantially that the input data (both static and dynamic, represented by the parameters) for every `p`'s step are being scanned the only time (by the single processing) during the *"specialization"* (the second) stage.

Online specialization performs metacompucation of the steps of the program `p` being transformed *"on the fly"* of analyzing various properties of the program; generally speaking, in no way a priori restricting itself both in any means and in the number of processing along the program `p` (or along segments (parts) of the program; for example, – along the input data of each program's step). Each such

[2] Here, in the task solved by S.A. Romanenko both `mix` must be replaced with `unmix`.

a processing gives a loop, which, in general case, can not be automatically recognized even if it works without any consequences, not doing transformations, but only looking for a property such that the program `p` does not satisfy the property. Hence, in general case, this loop will be presented in the residual program and will increasing the time complexity of the residual program. In any attempt of such self-application the data processings trying to separate (recognize) the static data from the dynamic ones will be observed by the transforming copy of the specializer. That essentially complicates its logic (as compared with the offline approach). Algorithmically undecidable and decidable parts of the logic are not separated at all.

Resume: development of the methods of online specialization is much more complicated as compared with the methods of offline specialization.

By definition, online specialization is lesser restricted in the methods being used than offline specialization and, as a consequence, is potentially considerably stronger. The most important point here is as follows: offline specialization is able to transform only the source program `p`, while online specialization is able to transform (and it really performs such transformations in the case of supercompilation) also *subprograms constructed by a specializer itself (and, hence, simple inefficient structures may be presented in such subprograms)*, but not only the `p`'s subprograms written by an human.

Supercompilation and "generalized partial computation" as collections of the online specialization methods provide much more stronger mechanisms for automatic program analyzing and transformation as compared with the partial evaluation technology. As a consequence, they set much more difficult problems: pure cognitive, algorithmic and technological (implementation of concrete specializers). The methods of supercompilation and generalized partial computation, unlike the partial evaluation methods, sometimes allows decrease the time complexity order of the programs being specialized. The residual programs are entirely constructed on the base of metainterpretation of the program being specialized, rather than on stepwise cleaning of the program. The section 4 is devoted to a survey of the supercompilation ideas.

Apparently, at the current moment, the ideas of generalized partial computation must be considered as the least developed. Unlike partial evaluation and supercompilation, the generalized partial computation approach to program specialization is not closed under itself. For example, an experimental semiautomatic specializer WSDFU announced in the 2002-nd [6] turns to an external theorem prover TPU [2] and to an external knowledge base for proving some properties of the programs being specialized. We have to note very interesting examples of specialization of programs with numerical arguments, generalized partial computation of which results in decreasing the time complexity of the programs [4,6]. Both the partial evaluation and supercompilation methods can do almost nothing with numerical data; here the main attention is attended to the programs transforming the binary trees (partial evaluation) and the finite sequences of *arbitrary* trees (supercompilation).

The partial evaluation methods as the simplest ones have been developed most thoroughly. Here the main contribution was made by N. D. Jones and his students. As we already mentioned above, the substantial part of the methods is the BTA-analysis approximating the algorithmically undecidable part of the specialization task *per se*. The task to be solved by the BTA is to most accurately recognize the control operators of the program being specialized, which can be evaluated without any information of the dynamic part of their input data, and at the same time the BTA has to terminate most frequently[3]. The most success in development of the BTA was achieved by means of analysis of a size change of a program being specialized (C. S. Lee, N. D. Jones, A. M. Ben-Amram [22]). The algorithms generalizing of parameterized configurations in supercompilation and generalized partial computation are analogues of the BTA; any preliminary annotation of the source program `p`, which is able to help in improving the generalization algorithms, can be very useful in the technologies. As far as we know there exist no attempts of using the BTA's methods in the supercompilation context. On the other hand, separation of the BTA and properly specialization, as well as the orientation on stepwise cleaning of the source program `p`, lead to direct (often undesirable) inheritance of the `p`'s properties by the residual program (see, for example, the Mogensen's paper [24]). Such an inheritance was the subject to be bypassed by S. A. Romanenko, when he was developing the arity rising algorithm (see above). The original restriction imposed on partial evaluation to construct the residual programs satisfying the properties formulated in the Task 1 (see above) puts irresistible difficulties on the way of very desirable optimizations. For example, the type specialization problem posed by N. D. Jones in [11] can be decided only in the frame of the Task 2. That was pointed by J. Hughes in the paper [10] describing some type specialization methods.

Speaking on partial evaluation the author have to mention the excellent N. D. Jones' book "Computability and Complexity from a Programming Perspective" (1997, [13]), in which N. D. Jones, with a purely theoretical viewpoint, tried to understand and generalize the experience accumulated in the partial evaluation filed.

4 An Historical Survey of Development of the Supercompilation Methods

In 1970-th years V. F. Turchin proposed a number of ideas on automatic program transformation. He called the idea as "supercompilation"[4].

[3] The requirement obligating a specializer to terminate on any its input data, usually, immediately confines the time measure function `T` to a simplest one; any interesting transformations cannot be expected.

[4] In our point of view the chosen name is poor. Supercompilation is not a kind of compilation; likewise a multivalued function is not a function and a vector field is not a field.

He posed a task to create tools for supervision over the operational semantics of a program, when the function F being calculated by the program is fixed. Such supervision must result in a new algorithmic definition of an *extension* of the function F. The new algorithm is constructed with the aim of quicker calculation of the F on fixed arguments (as compared with the original program).

Supercompilation was considered by V. F. Turchin with a point of view an application of his "metasystem transition philosophy". In the given paper we are not interested in the Turchin's philosophical constructions.

Below we name main stages of the history of development of the supercompilation ideas according to a Turchin's terminology given in the papers [43,44]. The supercompiler SCP4 was named by V. F. Turchin as well.

The first Turchin's publication "Ekvivalentnye preobrazovaniya rekursivnykh funktcij, opredelennykh na yazyke REFAL" (in Russian, "Equivalent transformations of recursive functions defined in REFAL") is dated with the 1972-nd [38]. The language REFAL was originally projected by V. F. Turchin as a metalanguage aiming to transform programs (in particularly, the programs written in the programming language REFAL). In this paper V. F. Turchin describes a fragment of REFAL called as *strict REFAL*, in which the time taken by matching of input data of a function with a pattern is uniformly bounded on size of the input data. To define the language fragment, a restriction was imposed on syntax of the patterns. The corresponding strict patterns were called as L-expressions. All models of the supercompilers[5] developed early than the supercompiler SCP4 used subject programming languages including only the strict patterns (or subsets of the strict pattern set). V. F. Turchin introduces (absolutely natural for any metacomputation) a concept of *driving* of the L-expressions, although he does not name the concept. He formulates an equivalent transformation calculus for the strict REFAL programs. The ideas of the calculus laid the basis for the supercompilation methods.

The Courant Computer Science report #20 stating many ideas on program transformation became the second important Turchin's work (1980, [40]), where many of the ideas are given very vaguely and, frequently, unconvincing. The work bristles with examples of non-algorithmic transformations and problem statements, most of which are not solved up to now. The examples substantially use the associative property of the REFAL's concatenation constructor. The report does put questions but does not answer the questions.

SCP1. The first simplest model of a supercompiler was implemented by V. F. Turchin, B. Nirenberg and D. V. Turchin in New York, in the 1981-st [48]. The supercompiler SCP1 was written in a REFAL's dialect. It worked in a dialog mode asking a human how to generalize the encountered configurations. Thus the main problem of approximation of the algorithmically undecidable part of the supercompilation logic was taken out of the consideration at all. The SCP1 represented an important step in polishing the driving algorithm, which performed metacomputation of calls by need (during the supercompilation stage).

[5] Including the supercompilers for LISP's toy-dialects.

The authors of the SCP1 succeeded in specialization of a number of simple examples. One of the examples became classical: a two-processing program replacing the symbol `'a'` with `'b'` in a given string and, after that, – the symbol `'b'` with `'c'` was specialized to an one-processing program (with respect to the call context of the two processings, which was directly represented by the syntactic composition $f(g(x))$). Thus the SCP1 was aimed to solve the second specialization task (see Section 2). Later such semantics was named as "lazy" semantics. In the 1990-th P. Wadler called a program transformation algorithm based on such driving as "deforestation" [50] and described an algorithmically *incomplete* language allowing only finitely many of parameterized configurations for a given program in iterative repetition of the lazy driving's steps.

SCP2 was developed by V. F. Turchin in the 1984-th. The Turchin's paper "The concept of a supercompiler" published in the 1986-th [42] and describing some ideas of the SCP2 implementation became the main classical work on supercompilation. The logical negation connective was introduced in the SCP2's language describing parameterized configurations. That allowed solve the following classical program transformation task by the supercompilation methods. An naive algorithm `p(s,x)` searching a substring `s` in a string `x` was transformed in an algorithm known as KMP [18]: by means of specialization of the source program with respect to the first argument $\texttt{p(s}_0\texttt{,x)}$. It was shown that the supercompiler can be used for automatic proofs of simple existence theorems. The generalization algorithm implemented in the SCP2 works ad hoc and, as a consequence, a human help is needed for the algorithm, if one wants to obtain more or less interesting transformations.

In the 1980-ths, at a Moscow REFAL workshop, A. Vedenov annotated a speedy completion (by himself) of a release of a REFAL supercompiler. Any publications or reports on such an actual implementation were not followed.

Two preprints written by Turchin's students were published by M. V. Keldysh Institute of Applied Mathematics of the Russian Academy of Sciences in the 1987-th. The works considered several supercompilation problems were "REFAL-4 – rasshirenie REFALa-2, obespechvajuschee vyrazimost' progonki" (S. A. Romanenko, in Russian, "REFAL-4 is an extension of REFAL-2, which supports expressibility of the driving", [32]) and "Metavychislitel' dlya yazyka REFAL, osnovnye ponyatiya i primery" (And. V. Klimov and S. A. Romanenko, in Russian, "A metaevaluator for the language REFAL, basic concepts and examples", [17]).

The work "The algorithm of generalization in the supercompiler" (1988, [41]) became the second impotent Turchin's paper. The paper describes an algorithm of generalization of function call's stacks and proves termination of the algorithm. The algorithm was called as the Obninsk algorithm cutting the stack; after a Russian city where Turchin presented the algorithm for the first time. The Obninsk algorithm is one of most important supercompilation algorithms. The supercompiler SCP2 was improved by the algorithm (see [46]).

The first actual attempt of self-application of a supercompiler was done in the 1989-th. A set of parameterized configurations of a program `p` being specialized is said to be a basic configurations' set if the `p` can be described in terms of the parameterized configurations. Finite sets of the basics configurations for a number of concrete simple tasks of self-application were manually constructed as a result of studying the trace of looping SCP2 self-application. These basic configurations' sets guaranteed termination of the supercompiler SCP2 running on the given tasks and not using any generalization algorithm. The generalization algorithm was withdrawn from the SCP2 with the goal to achieve self-application. The basic configurations' sets corresponding to the chosen simple self-application tasks were given as inputs to the SCP2. As a consequence, the SCP2 succeed in the self-application tasks. A paper describing these experiments saw light in the 1990-th [8]. Thus, the algorithmic decidable part (i.e. without the approximating generalization algorithm) of the simple specialization tasks *per se* was solved.

In the 1990-th N. V. Kondratiev [19], who is a Turchin's student, made an attempt of implementation of a supercompiler for REFAL. The attempt remains unfinished. A REFAL-graph language was used as an internal language for transformations. The REFAL-graph language is used in the supercompiler SCP4 (see below).

In the 1992-nd S. M. Abramov and R. F. Gurin (other Turchin's students) made a similar unfinished attempt for a simple model programming language working with the LISP data. The main hindrance, which they were not able to overcome, was development of an algorithm constructing output formats of the intermediate functions being constructed during supercompilation.

In the 1992-nd And. V. Klimov and R. Glück published a paper "Occam's razor in metacomputation: the notion of a perfect process tree" [7], where the driving algorithm described in the LISP terms (by means of binary trees). The main goal of the work was familiarization of western researchers with several simple ideas of supercompilation. The authors demonstrate the ideas on the simpler data as compared with the REFAL data. (For various reasons, importance of the associative property of the REFAL concatenation is not appreciated out of Russia until now.) The paper refined several concepts of the driving algorithm. A supercompiler for a simplest model LISP-like language was represented as well. This simple supercompiler also *a priory* assumes termination of the supercompilation process and does not use the principal generalization algorithm approximating the algorithmic undecidability of the specialization task *per se*.

SCP3. The experience of manual constructing the basic configuration's set in the experiments on self-application of the SCP2 made it clear that on the way of completely automatic self-application of supercompilers we are facing with many difficult problems. In the 1993-rd, V. F. Turchin decided to restrict the subject language of his supercompiler to a *"flat"* algorithmic complete fragment of REFAL-5. The fragment forbids the explicit syntactical constructions of the function call's compositions. The main goal in developing such a supercompiler transforming "flat" programs was achievement of its completely automatic

self-application. Here the SCP3 itself was developed in the terms of the whole REFAL. It was supposed that before self-application the SCP3's sources have to be translated in the flat REFAL. A crucial step in development of the supercompiler SCP3 was an extension of the parameter language describing the configurations of the program being transformed: adding new types of the parameters. The additional typing allows be more accurate in description of the self-application tasks (see details in [47,28]). In the 1994-th, it became possible to achieve completely automatic SCP3 self-application on a number of simple self-application tasks. Thus the long standing open question on the principal possibility of self-application of a specializer constructed on the base of the supercompilation methods was positively closed. In the 1996-th, V. F. Turchin, A. P. Nemytykh and V. A. Pinchuk published a paper ("A Self-Applicable Supercompiler", [29]) stating the basic ideas allowing to make these successful experiments and describing the experiments themselves. The algorithm generalizing the flat configurations still did not have a firm theoretical basis, although it did work completely automatically.

In the 1995-th, M. H. Sørensen (Denmark) [36] suggested to use an Higman-Kruskal relation [9,21] to make an important approximating decision by the generalization algorithm: "Given two configurations, have we to generalize them? Have not?". This suggestion put the algorithm generalizing the *"positive"* part of the configurations (that is to say, a part described without the negation connectivity) on a firm theoretical base. In the 1996-th, M. H. Sørensen, R. Glück and N. D. Jones published a paper [37] describing a model supercompiler for a simplest subset of the language LISP. In the supercompiler the language describing the parameterized configurations does not use the negation connective.

In the 1995-th, S. M. Abramov published a book "Metavychisleniya i ikh primenenie" (in Russian, "Metacomputation and their applications", [1]), in which the author describes an algorithm generalizing a negative part of the configurations. The negative part is given only in the unit-size terms (in the terms of *"symbols/atoms"*).

SCP4. A long-continued research (under supervision by V. F. Turchin) of the author (of this paper) resulted in development and implementation of an experimental supercompiler SCP4 (1999-2003) for a real programming language REFAL-5. In other words, without any restriction imposed on the language. Landmark program transformation algorithms were developed and implemented during this work. The very key algorithm from the series of the global analysis algorithms is an online algorithm constructing an output format of a function F. I.e. the output format is being constructed on the fly of the supercompilation process. That allows immediately use the constructed format for specialization (with respect the format) both other functions calling the F and the function F itself. The SCP4 is the first experimental free distributed supercompiler. An internet online version of the supercompiler is available as well. In the 2007-th, the author published a book "Superkompilyator SCP4: obschaya struktura" (in Russian, "The supercompiler SCP4: general structure") [28].

The supercompilation task is in essence a difficult task and, in its nature, an approximating task. Practically almost any interesting optimization problem is undecidable. The problem is, on one hand, in step-by-step movement to extension of the existing methods and algorithms and development of new ones; on the other hand, in compact description of the algorithms, which allows control the source code of the supercompiler itself. The existing collection of the basic methods used by the supercompiler SCP4 allows obtain enough interesting transformations thanks to diversity of composition of the methods. It is appropriate comparison here the situation with the classical Turing Machine, which possessing a collection of its trivial basic actions, nevertheless, allows define arbitrary algorithm by means of diversity of the elementary actions.

5 Supercompilation vs. Partial Evaluation

The Turing Machine (TM) gives another cause for returning to comparison supercompilation with partial evaluation.

What is the essence of the Jones' idea simplifying the online program transformation ideas and leading to partial evaluation? The answer is as follows. Given a finite collection of elementary program transformations $\{q_1, q_2, \ldots, q_n\}$ (i.e. a calculus) a supercompiler has to manipulate by the trivial transformations like a juggler with the goal of optimization of a given input program. One part of these trivial transformations (let it be $\{q_1, q_2, \ldots, q_m\}$) are responsible for generalization of the program configurations, while another part is directly used for metainterpretation (for specialization itself). As mentioned above (see Section 3) the BTA algorithm is an analogue of the generalization algorithm. The essence of the Jones's idea is to manipulate by the transformations $\{q_1, q_2, \ldots, q_m\}$ by means of the BTA only; and the result of such manipulation must be given as an input to the second transformation stage manipulating only the second part of the elementary transformations. Such a partition immediately leads to disastrous effects. To feel deeply what the effects are let us once again consider the TM example. Let us apply the Jones' idea to the basic TM's operators[6]

$$\{t_1, \ldots, \texttt{move}_{\texttt{to\_left}}, \texttt{move}_{\texttt{to\_right}}, \ldots, t_k\}$$

and part this collection in two ones:

$$\{t_1, \ldots, \texttt{move}_{\texttt{to\_left}}\} \text{ and } \{\texttt{move}_{\texttt{to\_right}}, \ldots, t_k\}.$$

Now according to partial evaluation we have to separately manipulate by the operators $\{t_1, \ldots, \texttt{move}_{\texttt{to\_left}}\}$ and just after that we are allowed to use the second part of the operators. Manipulation of the whole collection of the TM's operators provides possibility for generating any algorithm. But what can be programmed if we will follow the Jones' idea? The answer is trivial!

[6] Here $\texttt{move}_{\texttt{to\_left}}$ and $\texttt{move}_{\texttt{to\_right}}$ stand for the operators moving the TM's head.

References

1. Abramov, S. M.: Metacomputation and their applications (in Russian). 1995, Nauka-Fizmatlit, Moscow.
2. Chang, C., Lee, R. C.: Symbolic Logic and Mechanical Theorem Proving, 1973, Academic Press.
3. Futamura, Y.: Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. In: Systems. Computers. Controls. **2(5)** (1971) 45–50.
4. Futamura, Y., Nogi, K.: Generalized partial computation In the Proc. of the IFIP TC2 Workshop, (1988) 133–151. Amsterdam: North-Holland Publishing Co.
5. Futamura, Y., Nogi, K., Takano, A.: Essence of generalized partial computation. Theoretical Computer Science. **90** (1991) 61–79. Amsterdam. North-Holland Publishing Co.
6. Futamura, Y., Konishi, Z., Glück, R.: Program Transformation System Based on Generalized Partial Computation. New Generation Computing. Vol. **90** (2002) 75–99. Ohmsha Ltd. and Springer-Verlag.
7. Glück, R., Klimov, And. V.: Occam's razor in metacomputation: the notion of a perfect process tree. In Proc. of the Static Analysis Symposium, LNCS, Vol. **724** (1993) 112–123, Springer-Verlag.
8. Glück, R., Turchin, V. F.: Application of metasystem transition to function inversion and transformation. In the Proc. of the ISSAC'90 (1990), 286–287. ACM Press.
9. Higman, G.: Ordering by divisibility in abstract algebras. Proc. London Math. Soc. **2(7)** (1952) 326–336.
10. Hughes, J.: Type Specialization for the Lambda-Calculus; or a new Paradigm for Partial Evaluation Based on Type Inference. In Proc. the PEPM'96, LNCS, Vol. **1110** (1996) 183–215, Springer-Verlag.
11. Jones, N.D.: MIX ten years later. In the Proc. of the ACM SIGPLAN PEPM'95, (1995) 24–38, ACM Press.
12. Jones, N.D.: What not to do when writing an interpreter for specialization. In Proc. the PEPM'96, LNCS, Vol. **1110** (1996) 216–237, Springer-Verlag.
13. Jones, N. D.: Computability and Complexity from a Programming Perspective. (1997) The MIT Press.
14. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. (1993) Prentice Hall International.
15. Jones, N.D., Sestoft, P., Søndergaard, H.: An experiment in partial evaluation: the generation of a compiler generator. In Proc. of Conf. on Rewriting Techniques and Applications, LNCS, **202** (1985), 125–140. Springer-Verlag.
16. Jones, N.D., Sestoft, P., Søndergaard, H.: Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. Lisp and Symbolic Computation, **2(1)** (1989) 9–50.
17. Klimov, And. V., Romanenko, S. A.: A metaevaluator for the language REFAL, basic concepts and examples. (in Russian), Preprint Num. **71** (1987), M.V.Keldysh Institute of Applied Mathematics of Russian Academy of Sciences, Moscow.
18. Knuth, D. E., Morris, J. H., Pratt, V. R.: Fast Pattern Matching in strings. SIAM J. Comput., Vol. **6(2)** (1977) 323–350.
19. Kondratiev, N. V.: Approaches to construction of a supercompiler. (1990), (unpublished, private communication).
20. Korlyukov, A.V. User manual on the Supercompiler SCP4. (in Russian) (1999) `http://www.refal.net/supercom.htm`

21. Kruskal, J.B.: Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. Trans. Amer. Math. Society, **95** (1960) 210–225.
22. Lee, C. S., Jones, N. D., Ben-Amram, A. M.: The Size-Change Principle for Program Termination. ACM Symposium on Principles of Programming Languages. **28** (2001) 81–92, ACM press.
23. Leuschel, M.: Homeomorphic Embedding for Online Termination. In Proc. of the 8th Int. Workshop on Logic Program Synthesis and Transformation (LOPSTR), LNCS, Vol. **1559**, 199–218. Springer-Verlag.
24. Mogensen, T.: Evolution of Partial Evaluators: Removing Inherited Limits. In Proc. the PEPM'96 LNCS, Vol. **1110** (1996) 303–321, Springer-Verlag.
25. Nemytykh, A.P.: A Note on Elimination of Simplest Recursions.. In Proc. of the ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation, (2002) 138–146, ACM Press.
26. Nemytykh, A.P.: The Supercompiler SCP4: General Structure (extended abstract). In Proc. of the Perspectives of System Informatics, LNCS, **2890** (2003) 162–170, Springer-Verlag.
27. Nemytykh, A.P.: Playing on REFAL. In Proc. of the International Workshop on Program Understanding, (2003) 29–39, A.P. Ershov Institute of Informatics Systems, Syberian Branch of Russian Academy of Sciences. Accessible via: ftp://www.botik.ru/pub/local/scp/refal5/nemytykh_PU03.ps.gz
28. Nemytykh, A.P.: The Supercompiler SCP4: General Structure. Moscow, URSS. (A book in Russian, to appear)
29. Nemytykh, A. P., Pinchuk, V. A., Turchin, V. F.: A Self-Applicable Supercompiler. In Proc. the PEPM'96 LNCS, Vol. **1110** (1996) 322–337, Springer-Verlag. (`ftp://ftp.botik.ru/pub/local/APP/self-appl.ps.gz`).
30. Nemytykh, A.P., Turchin, V.F.: The Supercompiler SCP4: sources, on-line demonstration, `http://www.botik.ru/pub/local/scp/refal5/`, (2000).
31. Romanenko, S. A.: A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure (in Russian), Preprint Num. **20** (1987), M.V.Keldysh Institute of Applied Mathematics of Russian Academy of Sciences, Moscow.
32. Romanenko, S. A.: REFAL-4 is an extension of REFAL-2, which supports expressibility of the driving (in Russian), Preprint Num. **147** (1987), M.V.Keldysh Institute of Applied Mathematics of Russian Academy of Sciences, Moscow.
33. Romanenko, S. A.: A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In The Proc. of the IFIP TC2 Workshop, Partial Evaluation and Mixed Computation, (1988) 445–463, North-Holland Publishing Co.
34. Romanenko, S. A.: Arity raiser and its use in program specialization In the Proc. of the ESOP'90, LNCS, Vol. **432** (1990) 341–360, Springer-Verlag.
35. Sestoft, P.: The structure of a self-applicable partial evaluator. In the Proc. of the Programs as Data Objects, LNCS, Vol. **217** (1986) 236–256, Springer-Verlag.
36. Sørensen, M. H., Glück, R.: An algorithm of generalization in positive supercompilation. Logic Programming: Proceedings of the 1995 International Symposium (1995) 486–479, MIT Press.
37. Sørensen, M. H., Glück, R., Jones, N. D.: A positive supercompiler. Journal of Functional Programming, Vol. **6(6)** (1996) 811–838, MIT Press.
38. Turchin, V.F.: Equivalent transformations of recursive functions defined in REFAL. (in Russian), In the Proc. of the symposium on Theory of Languages and Methods of Constructing of Programming Systems, (1972) 31–42.

39. Turchin, V.F.: The use of metasystem transition in theorem proving and program optimization. In Proc. the 7th Colloquium on Automata, Languages and Programming, LNCS, Vol. **85** (1980) 645–657, Springer-Verlag.
40. Turchin, V.F.: The language Refal – The Theory of Compilation and Metasystem Analysis. Courant Computer Science Report, Num. **20** (February 1980), New York University.
41. Turchin, V.F.: The algorithm of generalization in the supercompiler. In the Proc. of the IFIP TC2 Workshop, (1988) 531–549.
42. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems. **8** (1986) 292–325, ACM Press.
43. Turchin, V.F.: Metacomputation: Metasystem transition plus supercompilation. In Proc. the PEPM'96, LNCS, Vol. **1110** (1996) 481–509, Springer-Verlag.
44. Turchin, V.F.: Supercompilation: Techniques and results. In the Proc. of PSI'96, LNCS, Vol. **1181** (1996) 227–248, Springer-Verlag.
45. Turchin, V.F.: Refal-5, Programming Guide and Reference Manual. Holyoke, Massachusetts. (1989) New England Publishing Co.
(electronic version: `http://www.botik.ru/pub/local/scp/refal5/` ,2000)
46. Turchin, V.F.: Metacomputation in the language Refal (1990). (unpublished, private communication)
47. Turchin, V. F., Nemytykh, A. P.: Metavariables: Their implementation and use in Program Transformation, Technical Report CSc. **TR 95-012** (1995), City College of the City University of New York.
48. Turchin, V. F., Nireberg, R., Turchin, D. V.: Experiments with a supercompiler Conference Record of the ACM Symposium on LISP and Functional Programming, (1982) 47–55, ACM Press.
49. Turchin, V.F., Turchin, D.V., Konyshev, A.P., Nemytykh, A.P.: Refal-5: sources, executable modules. `http://www.botik.ru/pub/local/scp/refal5/`, (2000)
50. Wadler, P.: Deforestation: Transforming programs to eliminate tree. Theoretical Computer Science, Vol. **73** (1990) 231–238.