The Essence of Program Transformation by Partial Evaluation and Driving*

Neil D. Jones

DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen, Denmark
neil@diku.dk

Abstract. An abstract framework is developed to describe program transformation by *specializing* a given program to a restricted set of inputs. Particular cases include partial evaluation [19] and Turchin's more powerful "driving" transformation [33]. Such automatic program speedups have been seen to give quite signifcant speedups in practical applications.
This paper's aims are similar to those of [18]: better to understand the fundamental mathematical phenomena that make such speedups possible. The current paper is more complete than [18], since it precisely formulates correctness of code generation; and more powerful, since it includes program optimizations not achievable by simple partial evaluation. Moreover, for the first time it puts Turchin's driving methodology on a solid semantic foundation which is not tied to any particular programming language or data structure.

This paper is dedicated to Satoru Takasu with thanks for good advice early in my career on how to do research, and for insight into how to see the essential part of a new problem.

1 Introduction

1.1 History

Automatic program specialization evolved independently at several different times and places [13,31,33,5,11,20]. In recent years partial evaluation has received much attention ([19,6], and several conferences), and work has been done on other automatic transformations including Wadler's well-known *deforestation* [36,7,26].

Many of these active research themes were anticipated in the 1970's by Valentin Turchin in Moscow [29,30] in his research on *supercompilation* (= supervised compilation), and experiments were made with implementations. Examples include program optimization both by deforestation and by partial evaluation; the use and significance of self-application for generating compilers and other program generators; and the use of grammars as a tool in program transformation [31,32,17]. Recent works on driving and supercompilation include [33,14,15,27,24,22,1].

* This work was supported in part by the Danish Natural Science Research Council (DART project) and by an Esprit Basic Research Action (Semantique).

1.2 Goals

The purpose of this paper is to formulate the essential concepts of supercompilation in an abstract and language-independent way. For simplicity we treat only imperative programs, and intentionally do not make explicit the nature of either commands or the store, except as needed for examples.

At the core of supercompilation is the program transformation called *driving* (Russian "progonka"). In principle driving is stronger than both deforestation and partial evaluation [27,36,12,19], and an example will be given to show this (the pattern matching example at the end of the paper). On the other hand, driving has taken longer to come into practical use than either deforestation or partial evaluation, for several reasons.

First, the greater strength of driving makes it correspondingly harder to tame; cause and effect are less easily understood than in deforestation and partial evaluation, and in fact it is only in the latter case that self-application has been achieved on practical applications. Second, the first papers were in Russian, and they and later ones used a computer language Refal[2] unfamiliar to western readers. Finally, the presentation style of the supercompilation papers is unfamiliar, using examples and sketches of algorithms rather than mathematical formulations of the basic ideas, and avoiding even set theory for philosophical reasons [34].

We hope the abstract framework will lead to greater practical exploitation of the principles underlying supercompilation (stronger program transformations, more automatic systems, new languages), and a better understanding in principle of the difficult problem of ensuring termination of program transformation.

1.3 Preliminary definitions

First, a quite abstract definition of an imperative program is given, as a state transition system. In our opinion the essence of the "driving" concept is more clearly exposed at this level. Later, a more intuitive flow chart formalism will be used for examples, and to clarify the problem of code generation.

Definition 1. An *abstract program* is a quadruple $\pi = (P, S, \rightarrow, p_0)$ where $p_0 \in P$ and $\rightarrow \subseteq (P \times S) \times (P \times S)$. Terminology: P is the set of *program points*, S is the set of *stores*, $\rightarrow$ is the *transition relation*, and p_0 is the *initial program point*. We write $\rightarrow$ in infix notation, e.g. $(p, s) \rightarrow (p', s')$ instead of $((p, s), (p', s')) \in \rightarrow$. A *state* is a pair $(p, s) \in P \times S$.

A store such as [`X`$\mapsto$ `1:2:[]`, `Y` $\mapsto$ `2:(4:5):[]`] usually maps program variables to their values. A program point may be a flow chart node, or can be thought of as a label in a program.

Definition 2. $p \in P$ is *one-way* if $(p, s_1) \rightarrow (p', s')$ and $(p, s_2) \rightarrow (p'', s'')$ imply $p' = p''$, i.e. there is at most one p' with $(p, \_) \rightarrow (p', \_)$. State (p, s) is *terminal* if

[2] Refal is essentially a language of Markov algorithms extended with variables. A program is a sequence of rewrite rules, used to transform data in the form of associative and possibly nested symbol strings. In contrast with most pattern matching languages, most general unifiers do not always exist.

$(p, s) \rightarrow (p', s')$ holds for no (p', s'). The abstract program π is *deterministic* if for all states (p, s), $(p, s) \rightarrow (p', s')$ and $(p, s) \rightarrow (p'', s'')$ imply $p' = p''$ and $s' = s''$.

Definition 3. A *computation* (from $s_0 \in S$) is a finite or infinite sequence

$$(p_0, s_0) \rightarrow (p_1, s_1) \rightarrow (p_2, s_2) \rightarrow \ldots$$

Notation: subsets of S will be indicated by overlines, so $\overline{s} \subseteq S$. Given this, and defining $\rightarrow^*$ to be the reflexive transitive closure of $\rightarrow$, the *input/output relation* that π defines on $\overline{s}_0 \subseteq S$ is

$$IO(\pi, \overline{s}_0) = \{(s_0, s_t) \mid s_0 \in \overline{s}_0, (p_0, s_0) \rightarrow^* (p_t, s_t), \text{ and } (p_t, s_t) \text{ is terminal}\}$$

More concretely, programs can be given by flow charts whose edges are labeled by commands. These are interpreted by a *command semantics*:

$$\mathcal{C}[\![\_]\!] : Command \rightarrow (S \stackrel{partial}{\longrightarrow} S)$$

where $Command$ and S are unspecified sets (but $S =$ the set of stores as above).

Definition 4. A *flow chart* is a rooted, edge-labeled directed graph $F = (P, E, p_0)$ where $p_0 \in P$ and $E \subseteq P \times Command \times P$ (the *edges* of F). We write $p \stackrel{C}{\Rightarrow} p'$ whenever $(p, C, p') \in E$.

If $p \stackrel{C}{\Rightarrow} p'$ then C denotes a store transformation, e.g. C could be an assignment statement changing a variable's value. The formulation includes tests too: the domain of partial function $\mathcal{C}[\![C]\!]$ is the set of stores which cause transition from program point p to p'. For example, command "`if odd(X) goto`" might label that edge, corresponding to "`p: if odd(X) then goto p'`" in concrete syntax.

Definition 5. The *program denoted by* F is $\pi^F = (P, S, \rightarrow, p_0)$, where

$$(p, s) \rightarrow (p', s') \text{ if and only if } s' = \mathcal{C}[\![C]\!]s \text{ for some } p \stackrel{C}{\Rightarrow} p'$$

2 Driven programs, without store transformations

A major use of driving (and partial evaluation) is for *program specialization*. For simplicity we begin with a rather weak form of driving that does not modify the store, and give a stronger version in the next section.

Given partial information about a program's inputs (represented by a subset $\overline{s}_0 \subseteq S$ of all possible stores), driving transforms program π into another program π_d that is equivalent to π on any initial store $s_0 \in \overline{s}_0$. The goal is efficiency: once π_d has been constructed, local optimizations of transition chain compression and reduced code generation can yield a much faster program than π, as seen in [18,19] and many others.

A useful principle is to begin by saying *what* is to be done, as simply as possible, before giving constructions and algorithms saying *how* it can be accomplished. We

thus first define what it means for a program π_d to be a "driven" form of program π, and defer the question of how to perform driving to Section 4.

Intuitively π_d is an "exploded" form of π in which any of π's program points p may have several annotated versions $(p, \overline{s}_1), (p, \overline{s}_2), \ldots$. Each $\overline{s}_i$ is a set of stores, required always to contain the current store in any computation by π_d.

Computations by π_d (state sequences) will be in a one-to-one correspondence with those of π, so nothing may seem to have been gained (and something lost, since π_d may be bigger than π). However, if control ever reaches an annotated program point $(p, \overline{s})$ in π_d, then the current runtime store *must lie in* $\overline{s}$. For example, $\overline{s}$ could be the set of all stores such that the value of variable X is always even.

This information is *the source of all improvements gained by partial evaluation or driving.* Its use is to optimize π_d by generating equivalent but more efficient code exploiting the information given by $\overline{s}$. In particular some computations may be elided altogether, since their effect can be achieved by using the $\overline{s}$ at transformation time; and knowledge of $\overline{s}$ often allows a much more economical representation of the stores $s \in \overline{s}$.

2.1 Abstract formulation

Definition 6. Given program $\pi = (P, S, \rightarrow, p_0)$, program $\pi_d = (P_d, S, \rightarrow_d, (p_0, \overline{s}_0))$ is an $\overline{s}_0$-*driven form* of π if $P_d \subseteq P \times \mathcal{P}(S)$ and π_d satisfies the following conditions.

1. $((p, \overline{s}), s) \rightarrow_d ((p', \overline{s}'), s')$ and $s \in \overline{s}$ imply $(p, s) \rightarrow (p', s')$. *soundness*
2. $(p, \overline{s}) \in P_d$, $(p, s) \rightarrow (p', s')$, and $s \in \overline{s}$ imply that there exists $\overline{s}'$ such that $((p, \overline{s}), s) \rightarrow_d ((p', \overline{s}'), s')$ *completeness*
3. $((p, \overline{s}), s) \rightarrow_d ((p', \overline{s}'), s')$ and $s \in \overline{s}$ imply $s' \in \overline{s}'$ *invariance* of $s \in \overline{s}$.

To begin with, $P_d \subseteq P \times \mathcal{P}(S)$, so a program point of π_d is a pair $(p, \overline{s})$ where $\overline{s} \subseteq S$ is a set of stores. The *soundness* condition says that π_d can do *only* the store transformations that π can do. The *completeness* condition says that for any driven program point $(p, \overline{s})$ of π_d, any store transformation that π can do from p on stores $s \in \overline{s}$ can also be done by π_d.

Programs may in principle be infinite, but in practice we are only interested in finite ones.

The significance of store sets. The invariance of $s \in \overline{s}$ in a transition $((p, \overline{s}), s) \rightarrow_d ((p', \overline{s}'), s')$ expresses a form of *information propagation* carried out at program transformation time [14,15].

One can think of a store set as a predicate describing variable value relationships, e.g. "X is even" or "$X = Y + 1 \wedge Z < Y$". Store sets could thus be manipulated in the form of logical formulas.

This view has much in common with regarding statements as forward or backward *predicate transformers*, as used by Dijkstra and many others for proving programs correct [10]. Further, a store set $\overline{s}$ that annotates a program point p corresponds to an *invariant*, i.e. a relationship among variable values that holds whenever control reaches point $(p, \overline{s})$ in the transformed program.

Instead of formulas, one could describe store sets using a set of *abstract values* Σ, using for example a function $\gamma : \Sigma \to \mathcal{P}(S)$ that maps an abstract value $\sigma \in \Sigma$ to the store set it denotes. In logic γ is called an *interpretation*, and Turchin uses the term *configuration* for such a store set description [33].

This idea is a cornerstone of abstract interpretation, where γ is called a *concretization function* [9,2,16]. Our approach can thus be described as *program specialization by abstract interpretation*. The abstract values are constructed 'on the fly' during program transformation to create new specialized program points. This is in contrast to most abstract interpretations, which iterate until the abstract values associated with the *original program*'s program points reach their collective least fixpoint.

Lemma 7. *If π_d is an $\overline{s}_0$-driven form of π, then for any $s_0 \in \overline{s}_0$ there is a computation*

$$(p_0, s_0) \to (p_1, s_1) \to (p_2, s_2) \to \ldots$$

if and only if there is a computation

$$((p_0, \overline{s}_0), s_0) \to ((p_1, \overline{s}_1), s_1) \to ((p_2, \overline{s}_2), s_2) \to \ldots$$

Proof. "If" follows from soundness, "only if" by completeness and invariance of $s \in \overline{s}$.

Corollary 8. $IO(\pi, \overline{s}_0) = IO(\pi_d, \overline{s}_0)$

Program specialization by driving. Informally, program π is transformed as follows:

1. Given π and an initial set of stores $\overline{s}_0$ to which π is to be specialized, construct a driven program π_d. In practice, π will be given in flow chart or other concrete syntactic form, and finite descriptions of store sets will be used.
2. Improve π_d by and removing unreachable branches, and by compressing sequences of one-way transitions

   $$((p, \overline{s}), s) \to ((p', \overline{s'}), s') \to \ldots \to ((p'', \overline{s''}), s'')$$

   into single-step transitions

   $$((p, \overline{s}), s) \to ((p'', \overline{s''}), s'')$$

3. If $\pi = \pi^F$ where F is a given flow chart, then F_d is constructed and improved in the same way: by compressing transitions, and generating appropriately simplified commands as edge labels.

The idea is that knowing a store set $\overline{s}$ gives contextual information used to transform π_d to make it run faster. Conditions for correct code generation will be given after we discuss the choice of store sets and the use of alternative store representations in Section 3.

2.2 Extreme and intermediate cases

In spite of the close correspondence between the computations of π and π_d, there is a wide latitude in the choice of π_d. Different choices will lead to different degrees of optimization. For practical use we need intermediate cases for which π_d has finitely many program points, and its store sets $\overline{s}$ are small enough (i.e. precise enough) to allow significant code optimization.

We will see a pattern-matching example where a program with two inputs of size m, n that runs in time $a \cdot m \cdot n$ can, by specializing to a fixed first input, be transformed into one running in time $b \cdot n$ where b is independent of m.

One extreme case is to choose every $\overline{s}$ to be equal to S. In this case π_d is identical to π, so no speedup is gained. Another extreme is to define π_d to contain $((p, \overline{s}), s) \rightarrow_d ((p', \{s'\}), s')$ whenever $(p, s) \rightarrow (p', s')$, $s \in \overline{s}$, and $(p, \overline{s}) \in P_d$. In this case π_d amounts to a totally unfolded version containing all possible computations on inputs from $\overline{s}_0$.

State set choice and code generation. The extreme just described will nearly always give infinite programs. It is not at all natural for code generation, as it deals with states one at a time.

In flow chart form, a test amounts to two different transitions $p \overset{C1}{\Rightarrow} p'$ and $p \overset{C2}{\Rightarrow} p''$ from the same p. A more interesting extreme can be obtained from the following principle: *the driven program should contain no tests that are not present in the original program.* The essence of this can be described without flow charts as follows.

Definition 9. *π_d requires no new tests* if whenever π contains $(p, s) \rightarrow (p', s')$, $s \in \overline{s}$, and π_d contains $((p, \overline{s}), s) \rightarrow_d ((p', \overline{s}'), s')$, then

$$\overline{s}' \supseteq \{s_2 \mid \exists s_1 \in \overline{s} \,.\, (p, s_1) \rightarrow (p', s_2) \text{ is in } \pi\}$$

This defines the new store set $\overline{s}'$ to be *inclusive*, meaning that it contains every store reachable from any store in $\overline{s}$ by π transitions from p to p'. The target store set $\overline{s}'$ of a driven transition $((p, \overline{s}), s) \rightarrow_d ((p', \overline{s}'), s')$ includes not only the target s' of s, but also the targets of all its "siblings" $s_1 \in \overline{s}$ that go from p to p'.

For deterministic programs, this amounts to requiring that π_d can only perform tests that are also performed by π. This is a reasonable restriction for code generation purposes, but is by no means necessary: if one somehow knows that the value of a given variable x must lie in a finite set $X = \{a, b, \ldots, k\}$, new tests could be generated to select specialized commands for each case of $x \in X$. See the discussion on 'bounded static variation' in [19].

An $\overline{s}_0$-driven form of π can always be obtained by choosing equality rather than set containment for $\overline{s}'$, and choosing π_d to contain the smallest set of program points including $(p_0, \overline{s}_0)$ and closed under the definition above. This extreme preserves all possible information about the computation subject to the inclusiveness condition. It can be used in principle to produce a "most completely optimized" version of the given program, but suffers from two practical problems:

First, this $\overline{s}_0$-driven π_d will very often contain infinitely many specialized program points $(p, \overline{s})$. Second, its transition relation may not be computable.

Generalization. It is a subtle problem in practice to guarantee that the transformed program both is finite and is more efficient than the original program. A solution in practice is not to work with the mathematically defined and usually infinite store sets above, but rather to use finite descriptions of perhaps larger sets $\overline{s}'' \supseteq \overline{s}'$ that can be manipulated by computable operations.

Finiteness of the transformed program can be achieved by choosing describable store sets that are larger than $\overline{s}'$ but which are still small enough to allow significant optimizations.

Turchin uses the term *configuration* for such a store set description, and *generalization* for the problem of choosing configurations to yield both finiteness and efficiency [33,35].

2.3 Driven flow charts

We now reformulate the former abstract definition for flow charts. For now we leave commands unchanged, as Section 3 will discuss store modifications and code generation together.

Definition 10. Given flow chart $F = (P, E, p_0)$ and $\overline{s}_0 \subseteq S$, $F_d = (P_d, E_d, (p_0, \overline{s}_0))$ is an $\overline{s}_0$-*driven form* of F if $P_d \subseteq P \times \mathcal{P}(S)$ and F_d satisfies the following conditions.

1. $(p, \overline{s}) \overset{C}{\Rightarrow} (p', \overline{s}')$ in F_d implies $p \overset{C}{\Rightarrow} p'$ in F *soundness.*
2. $(p, \overline{s}) \in P_d$, $\overline{s} \neq \{\}$, and $p \overset{C}{\Rightarrow} p'$ in F imply that $(p, \overline{s}) \overset{C}{\Rightarrow} (p', \overline{s}')$ in F_d for some $\overline{s}'$ *completeness.*
3. $(p, \overline{s}) \overset{C}{\Rightarrow} (p', \overline{s}')$ in F_d and $s \in \overline{s}$ and $s' = \mathcal{C}[\![C]\!]s$ is defined imply $s' \in \overline{s}'$ *invariance* of $s \in \overline{s}$.

Theorem 11. *If F_d is an $\overline{s}_0$-driven form of F, then π^{F_d} is an $\overline{s}_0$-driven form of π.*

Proof. This is easily verified from Definitions 5 and 10, as the latter is entirely parallel to Definition 6.

2.4 An example

Collatz' problem in number theory amounts to determining whether the following program terminates for all positive n. To our knowledge it is still unsolved.

A: **while** $n \neq 1$ **do**
  B: **if** n even
    then (C: $n := n \div 2$;)
    else (D: $n := 3 * n + 1$;)
  fi
  od
G:

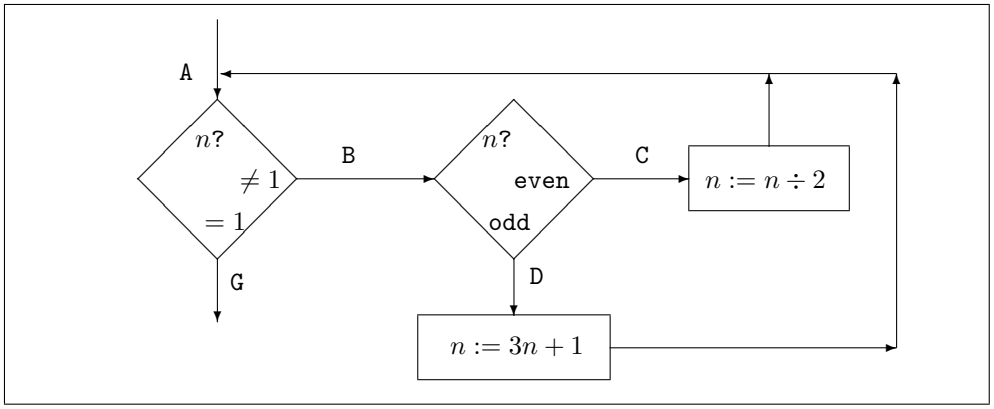


Figure 1: Diagram of a simple flow chart program

Its flow chart equivalent is $F = (P, E, 0)$ where $P = \{A, B, C, D, G\}$ and edge set E is given by the diagram in Figure 1. The program has only one variable n, so a store set is essentially a set of values.

We use just four store sets:

$$\begin{array}{ll} \mathit{Even} & = \{[n \mapsto x] \mid x \in \{0, 2, 4, \ldots\}\} \\ \mathit{Odd} & = \{[n \mapsto x] \mid x \in \{1, 3, 5, \ldots\}\} \\ \top & = \{[n \mapsto x] \mid x \in \mathcal{N}\} \\ \bot & = \{\} \end{array}$$

The flow chart F_d of Figure 2 is a driven version of F. Specialized program points $(D, \bot)$ and $(G, \bot)$ are unreachable since they have empty store sets. The driven version, though larger, contains two one-way transitions, from (A, Even) and (B, Even). (In the terminology of [32,15] these points are *imperfect*.) Transition compression redirects the branch from (D, Odd) to (C, Even) to give a somewhat better program, faster in that two tests are avoided whenever n becomes odd.

3 Driven programs, with store transformations

According to Definition 6, a driven program π_d has exactly the same stores as π. As a consequence the only real optimizations that can occur are from collapsing one-way transition chains, and no real computational saving happens. We now revise this definition, "retyping" the store to obtain more powerful transformations such as those of partial evaluation by projections [19,18,21] or arity raising [25].

3.1 Abstract formulation

From now on S_d will denote the set of possible stores in driven program π_d. Given the knowledge that $s \in \overline{s}$, a store s of π can often be represented in the driven program π_d by a simpler store $s_d \in S_d$. For example, if

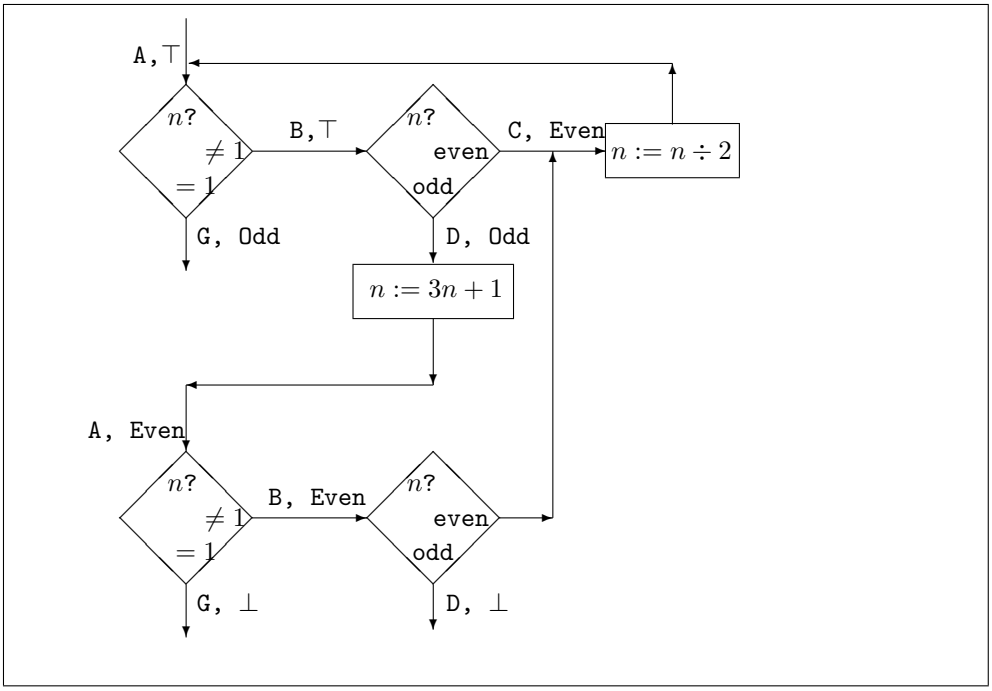


Figure 2: A driven version of the same program

$$\overline{s} = \{\ [\texttt{X} \mapsto 1,\ \texttt{Y} \mapsto y,\ \texttt{Z} \mapsto 3] \mid y \in \mathcal{N}\}$$

then $s \in \overline{s}$ at π_d program point $(p, \overline{s})$ can be represented by the value of `Y` alone since `X`, `Z` are known from context. In practice, $\overline{s}$ will be described finitely, e.g. by an abstract value σ in description set Σ:

$$\sigma = [\texttt{X} \mapsto 1,\ \texttt{Y} \mapsto \top,\ \texttt{Z} \mapsto 3].$$

together with concretization function (or interpretation) $\gamma : \Sigma \to \mathcal{P}(S)$. To formalize this abstractly, we assume given a function

$$\Delta : \mathcal{P}(S) \times S_d \stackrel{partial}{\longrightarrow} S$$

satisfying the following two properties (note that Δ is written in infix notation.):

1. $\overline{s}\Delta s_d \in \overline{s}$ whenever $\overline{s} \subseteq S, s_d \in S_d$, and $\overline{s}\Delta s_d$ is defined; and
2. $\overline{s}_0\Delta s_d = \overline{s}_1\Delta s'_d = s$ implies $s_d = s'_d$

One can think of Δ as a reconstruction function to build s from store set $\overline{s}$ and a driven store s_d. For example, if $\overline{s}$ is as above and if s_d is, say, $[Y \mapsto 5]$ then we would have $\overline{s}\Delta s_d = [X \mapsto 1, Y \mapsto 5, Z \mapsto 3]$.

The restriction $\overline{s}\Delta s_d \in \overline{s}$ says that s_d can only represent a store in the current $\overline{s}$. The second restriction says that Δ is injective in its second argument.

The previous formulation without store transformations is expressible by putting $S = S_d$, and letting $\overline{s}\Delta s_d = s_d$ when $s_d = s_d \in \overline{s}$, with $\overline{s}\Delta s_d$ undefined otherwise.

We will see that allowing alternative representations of the driven stores enables much stronger program optimizations. The new Definition 6 is as follows. The essential idea is that a transition

$$(p, s) \to (p', s') \quad = \quad (p, \overline{s}\Delta s_d) \to (p', \overline{s}'\Delta s'_d)$$

is transformed, by a kind of reassociation, into a specialized transition of form

$$((p, \overline{s}), s_d) \to_d ((p', \overline{s}'), s'_d)$$

Definition 12. Program $\pi_d = (P_d, S_d, \to_d, (p_0, \overline{s}_0))$ is an $\overline{s}_0$-*driven form* of $\pi = (P, S, \to, p_0)$ in case $P_d \subseteq P \times \mathcal{P}(S)$ and π_d satisfies the following conditions.

1. $((p, \overline{s}), s_d) \to_d ((p', \overline{s}'), s'_d)$ implies $s = \overline{s}\Delta s_d$ and $s' = \overline{s}'\Delta s'_d$ for some s, s', and $(p, s) \to (p', s')$. *soundness*

2. $(p, \overline{s}) \in P_d$, $s \in \overline{s}$, and $(p, s) \to (p', s')$ imply there are $s_d, s'_d, \overline{s}'$ such that $s = \overline{s}\Delta s_d$, $s' = \overline{s}'\Delta s'_d$, and $((p, \overline{s}), s_d) \to_d ((p', \overline{s}'), s'_d)$. *completeness*

3. $((p, \overline{s}), s_d) \to_d ((p', \overline{s}'), s'_d)$ imply $\overline{s}'\Delta s'_d \in \overline{s}'$ *invariance* of $s \in \overline{s}$

Condition 3 is actually redundant, as it follows from 1 and the requirement on Δ.

Lemma 13. *If π_d is an $\overline{s}_0$-driven form of π, then for any computation*

$$(p_0, s_0) \to (p_1, s_1) \to (p_2, s_2) \to \ldots$$

with $s_0 = \overline{s}_0\Delta s_{0d}$ there is a computation

$$((p_0, \overline{s}_0), s_{d0}) \to_d ((p_1, \overline{s}_1), s_{d1}) \to_d ((p_2, \overline{s}_2), s_{d2}) \to_d \ldots$$

with $s_i = \overline{s}_i\Delta s_{di}$ for all i. Further, for any such π_d computation with $s_0 = \overline{s}_0\Delta s_{d0}$, there is a corresponding π computation with $s_i = \overline{s}_i\Delta s_{di}$ for all i.

The first part follows from initialization and completeness, and the second by soundness and invariance. The corollary on equivalent input/output behaviour requires a modification.

Corollary 14. *If every $s_0 \in \overline{s}_0$ equals $\overline{s}_0\Delta s_{0d}$ for some s_{0d}, then $IO(\pi, \overline{s}_0) =$*

$$\{(\overline{s}_0\Delta s_{0d}, \overline{s}\Delta s_d) \mid \overline{s}_0\Delta s_{0d} \in \overline{s}_0 \text{ and } (((p_0, \overline{s}_0), s_{0d}), ((p, \overline{s}), s_d)) \in IO(\pi_d, \overline{s}_{0d})\}$$

3.2 Correctness of code in driven flow charts

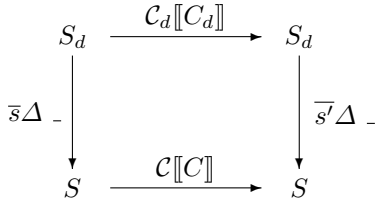
We now redefine driven flow charts to allow different code in F_d than in F. Commands labeling edges of F_d will be given subscript d. Their semantic function is:

$$\mathcal{C}_d[\![\_]\!] : Command_d \to (S_d \overset{partial}{\longrightarrow} S_d)$$

The following rather technical definition can be intuitively understood as saying that for each paired $p \overset{C}{\Rightarrow} p'$ and $(p, \overline{s}) \overset{C_d}{\Rightarrow} (p', \overline{s}')$, the diagram corresponding to equation

$$\mathcal{C}[\![C]\!](\overline{s} \Delta s_d) = \overline{s}' \Delta (\mathcal{C}_d[\![C_d]\!] s_d)$$

commutes, provided that various of its subexpressions are defined.



Definition 15. Given flow chart $F = (P, E, p_0)$ and $\overline{s}_0 \subseteq S$, $F_d = (P_d, E_d, (p_0, \overline{s}_0))$ is an $\overline{s}_0$-*driven form* of F if $P_d \subseteq P \times \mathcal{P}(S)$ and F_d satisfies the following conditions.

1. For each $(p, \overline{s}) \overset{C_d}{\Rightarrow} (p', \overline{s}') \in E_d$ there exists $p \overset{C}{\Rightarrow} p' \in E$ such that $s = \overline{s} \Delta s_d$ and $s' = \mathcal{C}[\![C]\!]s$ are defined if and only if $s'_d = \mathcal{C}_d[\![C_d]\!] s_d$ and $s' = \overline{s}' \Delta s'_d$ are defined *soundness*
2. If $p \overset{C}{\Rightarrow} p'$, $(p, \overline{s}) \in P_d$, and both $s = \overline{s} \Delta s_d$ and $s' = \mathcal{C}[\![C]\!]s$ are defined, then F_d has an edge $(p, \overline{s}) \overset{C_d}{\Rightarrow} (p', \overline{s}')$ such that $s' = \overline{s}' \Delta (\mathcal{C}_d[\![C_d]\!] s_d)$ *completeness*
3. $(p, \overline{s}) \overset{C_d}{\Rightarrow} (p', \overline{s}')$, $p \overset{C}{\Rightarrow} p'$, and both $s = \overline{s} \Delta s_d$ and $s' = \mathcal{C}[\![C]\!]s$ are defined imply $\mathcal{C}_d[\![C_d]\!] s_d \in \overline{s}'$ *invariance* of $s \in \overline{s}$.

Theorem 16. *If F_d is an $\overline{s}_0$-driven form of F, then π^{F_d} is an $\overline{s}_0$-driven form of π^F.*

Proof. This is easily verified from Definitions 5 and 15, as the latter is entirely parallel to Definition 12.

3.3 Partial evaluation by projections

Suppose there is a way to decompose or factor a store s into static and dynamic parts without loss of information (a basic idea in [18,19]). A *data division* is a triple of functions $(stat : S \to S_s, dyn : S \to S_d, pair : S_s \times S_d \to S)$. The ability to decompose and recompose without information loss can be expressed by three equations:

$$\begin{aligned} pair(stat(s), dyn(s)) &= s \\ stat(pair(v_s, v_d)) &= v_s \\ dyn(pair(v_s, v_d)) &= v_d \end{aligned}$$

An example. For example, a division could be given (as in [18,19]) by an $S - D$ vector, for instance SDD specifies the division of $S = \mathcal{N}^3$ into $\mathcal{N} \times \mathcal{N}^2$ where $pair(n,(x,a)) = (n,x,a)$, $stat(n,x,a) = n$, and $dyn(n,x,a) = (x,a)$. Using this, the program

$$\begin{aligned} f(n,x) &= g(n,x,1) \\ g(n,x,a) &= \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } g(n-1,x,x*a) \end{aligned}$$

can be specialized with respect to known $n = 2$ to yield:

$$\begin{aligned} f_2(x) &= g_2(x,1) \\ g_2(x,a) &= g_1(x,x*a) \\ g_1(x,a) &= g_0(x,x*a) \\ g_0(x,a) &= 1 \end{aligned}$$

which by transition compression can be further reduced to

$$f_2(x) = x * x$$

Relationship between driving and projections. This method can be interpreted in current terms as specialization by using store sets that are equivalence classes with respect to static projections, i.e. every store set is of the following form for some $v_s \in S_s$:

$$\overline{s}_{v_s} = \{s \mid stat(s) = v_s\}$$

Store reconstruction can be expressed by defining: $\overline{s}_{v_s} \Delta v_d = pair(v_s, v_d)$. A specialized program π_d in [18,19] only contains transitions of form

$$((p, stat(s)), dyn(s)) \rightarrow ((p', stat(s')), dyn(s'))$$

where π contains $(p,s) \rightarrow (p',s')$. This corresponds to our soundness condition. The set "poly" in [18,19]) is constructed so if $(p_0,s_0) \rightarrow^* (p,s)$ by π for some $s_0 \in \overline{s}_0$, then poly and so π_d contains a specialized program point $(p, stat(s))$, ensuring completeness. Invariance of $s \in \overline{s}$ is immediate since every specialized state is of the form $((p, \overline{s}_{v_s}), v_d)$, and

$$\overline{s}_{v_s} \Delta v_d = pair(v_s, v_d) \in \{s \mid stat(s) = v_s\}$$

since $stat(pair(v_s, v_d)) = v_s$. The following definition is central in [18,19]:

Definition 17. Function $stat : S \rightarrow S_d$ is *congruent* if for any π transitions $(p,s) \rightarrow (p',s')$ and $(p,s_1) \rightarrow (p',s'_1)$, if $stat(s) = stat(s_1)$, then $stat(s') = stat(s'_1)$.

This is essentially the "no new tests" requirement of Definition 9.

4 An algorithm for driving

The driving algorithm of Figure 3 manipulates store descriptions $\sigma \in \Sigma$, rather than store sets. For the x^n example above, Σ is the set of all store descriptions σ of the form

$$\sigma = [n \mapsto u, x \mapsto \top, a \mapsto \top]$$

where $u \in \mathcal{N}$. We assume given a *concretization function* $\gamma : \Sigma \to \mathcal{P}(S)$ defining their meanings, and that the test "is $\gamma\sigma = \{\}$?" is computable, i.e. that we can recognize a description of the empty set of stores.

In addition we assume given a *store set update* function

$$\mathcal{S} : Command \times \Sigma \to \Sigma$$

and a *code generation* function

$$\mathcal{G} : Command \times \Sigma \to Command_d$$

Correctness criterion. For any $C \in Command, \sigma \in \Sigma, s_d \in S_d$, let $\sigma' = \mathcal{S}(\sigma, C)$ and $C_d = \mathcal{G}(\sigma, C)$. Definition 15 requires $\mathcal{C}[\![C]\!](\gamma\sigma \Delta s_d) = (\gamma\sigma')\Delta(\mathcal{C}_d[\![C_d]\!]s_d)$ under certain conditions (where $t = t'$ means both are defined and the values are equal):

1. $s = (\gamma\sigma)\Delta s_d$ and $s'_d = \mathcal{C}_d[\![C_d]\!]s_d$ imply $\mathcal{C}[\![C]\!]s = (\gamma\sigma')\Delta s'_d$ — *soundness*
2. $s' = \mathcal{C}[\![C]\!]s$ and $s = (\gamma\sigma)\Delta s_d$ imply $s' = (\gamma\sigma')\Delta(\mathcal{C}_d[\![C_d]\!]s_d)$ — *completeness*
3. $s = (\gamma\sigma)\Delta s_d \in \gamma\sigma$ implies $\mathcal{C}_d[\![C_d]\!]s_d \in \gamma\sigma'$ — *invariance* of $s \in \overline{s}$.

4.1 Example: pattern matching in strings

A way to test a program transformation method's power is to see whether it can derive certain well-known efficient programs from equivalent naive and inefficient programs. One of the most popular of such tests is to generate, from a naive pattern matcher and a fixed pattern, an efficient pattern matcher as produced by the Knuth-Morris-Pratt algorithm. We shall call this *the KMP test* [27].

First we give a program for string pattern matching.

$$\begin{aligned}
&match\ p\ s && = loop\ p\ s\ p\ s\\
&\\
&loop\ []\ ss\ op\ os && = True\\
&loop\ (p : pp)\ []\ op\ os && = False\\
&loop\ (p : pp)\ (s : ss)\ op\ os && = \textbf{if}\ p = s\ \textbf{then}\ loop\ pp\ ss\ op\ os\ \textbf{else}\ next\ op\ os\\
&\\
&next\ op\ [] && = False\\
&next\ op\ (s : ss) && = loop\ op\ ss\ op\ ss
\end{aligned}$$

For conciseness in exposition, we specify the store sets that are encountered while driving *match AAB u* by means of terms containing free variables. These are assumed to range over all possible data values. Given this, the result of driving can

```
read F = (P, E, p0);
read σ0;
Pending := {(p0, σ0)};                    (* Unprocessed program points *)
SeenBefore := {};                         (* Already processed pgm. points *)
Pd := {(p0, σ0)};                         (* Initial program points *)
Ed := {};                                 (* Initial edge set *)
while ∃(p, σ) ∈ Pending do                (* Choose an unprocessed point *)
  Pending := Pending \ {(p, σ)};
  SeenBefore := SeenBefore ∪ {(p, σ)};
  forall p ⇒^C p' ∈ E do                  (* Scan all transitions from p *)
    σ' := S(σ, C);                        (* Update store set description *)
    if γσ' ≠ {} then                      (* Generate code if nontrivial *)
      Pd := Pd ∪ {(p', σ')};
      if (p', σ') ∉ SeenBefore then add (p', σ') to Pending;
      Cd := G(σ, C);                      (* Generate code *)
      Add edge (p, σ) ⇒^Cd (p', σ') to Ed; (* Extend flow chart by one edge *)
Fd := (Pd, Ed, (p0, σ0));
```

Figure 3: An algorithm for driving

be described by the configuration graph seen in the Figure ending this paper (where some intermediate configurations have been left out). More details can be seen in [27].

The program generated is:

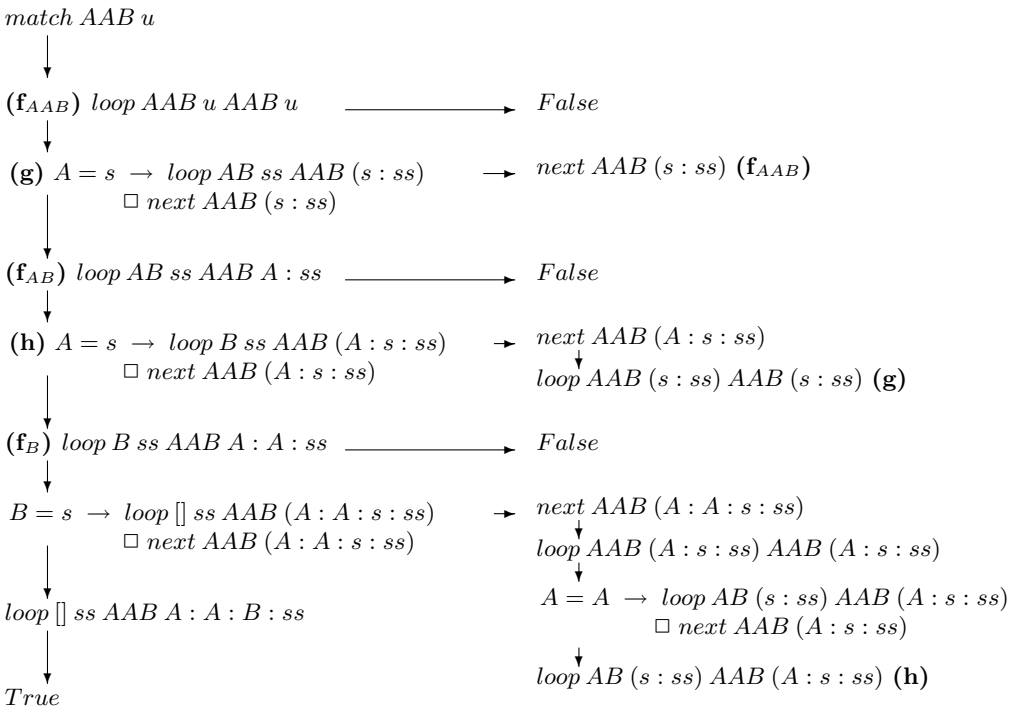
$$
\begin{array}{ll}
f\ u & = f_{AAB}\ u \\
 & \\
f_{AAB}\ [] & = False \\
f_{AAB}\ (s : ss) & = g\ s\ ss \\
 & \\
g\ s\ ss & = \textbf{if}\ A = s\ \textbf{then}\ f_{AB}\ ss\ \textbf{else}\ f_{AAB}\ ss \\
 & \\
f_{AB}\ [] & = False \\
f_{AB}\ (s : ss) & = h\ s\ ss \\
 & \\
h\ s\ ss & = \textbf{if}\ A = s\ \textbf{then}\ f_{B}\ ss\ \textbf{else}\ g\ ss \\
 & \\
f_{B}\ [] & = False \\
f_{B}\ (s : ss) & = \textbf{if}\ A = s\ \textbf{then}\ g\ s\ ss\ \textbf{else} \\
 & \quad \textbf{if}\ B = s\ \textbf{then}\ true\ \textbf{else}\ h\ s\ ss
\end{array}
$$

This is in essence a KMP pattern matcher, so driving passes the KMP test. It is interesting to note that driving has transformed a program running in time $O(m \cdot n)$

into one running in time $O(n)$, where m is the length of the pattern and n is the length of the subject string.

Using configurations as above can result in some redundant tests, because we only propagate positive information (what term describes the negative outcome of a test?). However this problem can easily be overcome by using both positive and negative environments, see [15].

Partial evaluators of which we know cannot achieve this effect without nontrivial human rewriting of the matching program.



4.2 Finiteness and generalization

Σ is usually an infinite set, causing the risk of generating infinitely many different configurations while driving. Turchin uses the term *generalization* for the problem of choosing configurations to yield both finiteness and efficiency [33,35].

The idea is to choose elements $\sigma' = \mathcal{S}(\sigma, C)$ which are "large enough" to ensure finiteness of the transformed program, but are still small enough to allow significant optimizations. This may require one to *ignore some information* that is available at transformation time, i.e. to choose descriptions of larger and so less precise store sets than would be possible on the basis of the current σ and C.

How to achieve termination without overgeneralization is not yet fully understood. Turchin advocates an online technique, using the computational history of the driving process to guide the choices of new σ' [35]. It is as yet unclear whether self-application for practical compiler generation can be achieved in this way, or whether some form of preprocessing will be needed. If offline preprocessing is needed, it will certainly be rather different from "binding-time analysis" as used in partial evaluation [19].

Acknowledgement

Many useful comments on this paper were made by Patrick Cusot, Robert Glück, Andrei Klimov, Sergei Romanenko, Morten Heine Sørensen, and Carolyn Talcott.

References

1. Sergei M. Abramov, Metacomputation and program testing. In: *1st International Workshop on Automated and Algorithmic Debugging*. (Linköping, Sweden). pp. 121-135, Linköping University 1993.
2. Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.
3. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
4. Lennart Augustsson, Compiling lazy pattern-matching. *Conference on Functional Programming and Computer Architecture*, ed. J.-P. Jouannoud. Lecture Notes in Computer Science 201, Springer-Verlag, 1985.
5. L. Beckman et al. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4), pp. 319–357, 1976.
6. D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop*. North-Holland, 1988. 625 pages.
7. Wei-Ngan Chin, Safe fusion of functional expressions II: further improvements. *Journal of Functional Programming*. To appear in 1994.
8. Charles Consel and Olivier Danvy, Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30, pp. 79–86, January 1989.
9. Patrick Cousot and Radhia Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles on Programming Languages*, pp. 238–252, New York: ACM Press, 1977.
10. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
11. Andrei P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18, pp. 41–67, 1982.
12. Alex B. Ferguson and Philip Wadler, When will deforestation stop? *Glasgow Workshop on Functional Programming*, August 1988.
13. Yoshihiko Futamura and Kenroku Nogi, Generalized partial computation. In *Partial Evaluation and Mixed Computation,* Eds. A. P. Ershov, D. Bjørner and N. D. Jones, North-Holland, 1988.
14. Robert Glück and Valentin F. Turchin, Application of metasystem transition to function inversion and transformation. *Proceedings of the ISSAC '90*, pp. 286-287, ACM Press,1990.

15. Robert Glück and Andrei V. Klimov, Occam's razor in metacomputation: the notion of a perfect process tree. In *Static analysis Proceedings*, eds. P. Cousot, M. Falaschi, G. Filé, G. Rauzy. Lecture Notes in Computer Science 724, pp. 112-123, Springer-Verlag, 1993.
16. Neil D. Jones and Flemming Nielson, Abstract interpretation: a semantics-based tool for program analysis, 122 pages. In *Handbook of Logic in Computer Science*, Oxford University Press to appear in 1994.
17. Neil D. Jones, Flow analysis of lazy higher-order functional programs. In *Abstract Interpretation of Declarative Languages*, pp. 103-122. Ellis Horwood, 1987.
18. Neil D. Jones, Automatic program specialization: A re-examination from basic principles, in D. Bjørner, A.P. Ershov, and N.D. Jones (eds.), *Partial Evaluation and Mixed Computation*, pp. 225–282, Amsterdam: North-Holland, 1988.
19. Neil D. Jones, Carsten Gomard and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 425 pp., 1993.
20. Stephen S. Kleene, *Introduction to Metamathematics.* Van Nostrand, 1952, 550 pp.
21. John Launchbury, *Projection Factorisations in Partial Evaluation.* Cambridge: Cambridge University Press, 1991.
22. Andrei V. Klimov and Sergei Romanenko, A metaevaluator for the language Refal: basic concepts and examples. Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow. Preprint No. 71, 1987 (in Russian).
23. Donald E. Knuth, James H. Morris, and Vaughan R. Pratt, Fast pattern matching in strings, *SIAM Journal of Computation*, 6(2), pp. 323–350, 1977.
24. Alexander Y. Romanenko, The generation of inverse functions in Refal, in D. Bjørner, A.P. Ershov, and N.D. Jones (eds.), *Partial Evaluation and Mixed Computation*, pp. 427-444, Amsterdam: North-Holland, 1988.
25. Sergei A. Romanenko, A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjørner, A.P. Ershov, and N.D. Jones (eds.), *Partial Evaluation and Mixed Computation*, pp. 445–463, Amsterdam: North-Holland, 1988.
26. Morten Heine Sørensen, *A grammar-based data flow analysis to stop deforestation.Colloquium on Trees and Algebra in Programming (CAAP)*, edinburgh, Scotland. Lecture Notes in Computer Science, Springer-Verlag, to appear in 1994.
27. Morten Heine Sørensen, Robert Glück and Neil D. Jones, Towards unifying partial evaluation, deforestation, supercompilation, and GPC. *European Symposium on Programming (ESOP).* Lecture Notes in Computer Science, Springer-Verlag, to appear in 1994.
28. Akihiko Takano, Generalized partial computation for a lazy functional language. *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, eds. Neil D. Jones and Paul Hudak, ACM Press, 1991.
29. Valentin F. Turchin, Equivalent transformations of recursive functions defined in Refal. In: *Teorija Jazykov i Metody Programmirovanija* (Proceedings of the Symposium on the Theory of Languages and Programming Methods). (Kiev-Alushta, USSR). pp. 31-42, 1972 (in Russian).
30. Valentin F. Turchin, Equivalent transformations of Refal programs. In: *Avtomatizirovannaja Sistema upravlenija stroitel'stvom.* Trudy CNIPIASS, 6, pp. 36-68, 1974 (in Russian).
31. Valentin F. Turchin, *The Language Refal, the Theory of Compilation and Metasystem Analysis.* Courant Computer Science Report 20, 245 pages, 1980.
32. Valentin F. Turchin, Semantic definitions in Refal and automatic production of compilers. *Semantics-Directed Compiler Generation*, Aarhus, Denmark. Lecture Notes in Computer Science, Springer-Verlag, pp. 441-474, vol. 94, 1980.

33. Valentin F. Turchin, The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3), pp. 292–325, July 1986.
34. Turchin V. F., A constructive interpretation of the full set theory. In: *The Journal of Symbolic Logic*, 52(1): 172-201, 1987.
35. Valentin F. Turchin, The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones (eds.), *Partial Evaluation and Mixed Computation*, pp. 531-549, Amsterdam: North-Holland, 1988.
36. Philip L. Wadler, Deforestation: transforming programs to eliminate trees. European Symposium On Programming (ESOP). Lecture Notes in Computer Science 300, pp. 344-358, Nancy, France, Springer-Verlag, 1988.