Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree

Robert Glück[1] Andrei V. Klimov[2]

Institut für Computersprachen
University of Technology Vienna
A-1040 Vienna, Austria

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
125047 Moscow, Russia

Abstract. We introduce the notion of a perfect process tree as a model for the full propagation of information in metacomputation. Starting with constant propagation we construct step-by-step the driving mechanism used in supercompilation which ensures the perfect propagation of information. The concept of a simple supercompiler based on perfect driving coupled with a simple folding strategy is explained. As an example we demonstrate that specializing a naive pattern matcher with respect to a fixed pattern obtains the efficiency of a matcher generated by the Knuth, Morris & Pratt algorithm.

1 Introduction

Research in the field of program specialization extends the state-of-the-art in two directions: extending existing methods to new languages and improving the techniques for the specialization of programs. While the first goal can be stated clearly, the second goal is often expressed in rather vague terms such as 'strength' and 'transformation depth'. Often new methods are introduced rather ad hoc and it is hard to see how much has been achieved and what the limitations are.

How should one assess the quality of a program specialization method? Various criteria are conceivable. In this paper we propose the notion of a *perfect process tree*. The goal is to propagate 'enough' information to be able to prune all infeasible program branches at specialization time. Many existing methods, such as partial evaluation, develop imperfect process trees. This should not be taken as a negative statement, but — on the contrary — as a motivation for improving specialization methods further. We present different methods for specializing a simple programming language with tree-structured data, called S-Graph. Starting from constant propagation we develop step-by-step a driving mechanism which ensures the perfect propagation of information along the specialized program branches. The use of perfect driving is shown by introducing a supercompiler with a simple folding strategy. As an example we demonstrate that specializing a naive pattern matcher with respect to a fixed pattern obtains the efficiency of a matcher generated by the Knuth, Morris & Pratt algorithm.

2 Background: Process-Based Transformation

In this section we review Turchin's concept of process-based program transformation. A program $p \in Pgm$, $p : Data \rightarrow Data$, given data $d \in Data$, defines the behavior of a machine: a *computation process*. A computation process is a potentially infinite sequence of states and transitions. A state may contain a program point and a store. Each state and transition in a deterministic computation process is fully defined: $process(p, d) = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \ldots$ (This may be defined by some well-understood semantics such as operational or denotational semantics, omit-

[1] Supported by the Austrian Science Foundation (FWF) under grant number J0780-PHY.
Current address: DIKU, Dept. of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark. Email: glueck@diku.dk.
[2] Supported by the Russian Foundation for Fundamental Research under grant number 93-12-628 and in part by the 'Österreichische Forschungsgemeinschaft' under grant number 06/1789.
Email: And.Klimov@refal.msk.su.

ted here). The *set of processes*, $P = \{ process(p, d) \mid d \in Data \}$, captures the semantics of a program as a whole. But how can one describe and manipulate the set of processes constructively?

Process Graph. For process-based program transformation one instead uses a *process graph*. A process graph is used to describe and manipulate the set of computation processes. In the following this will be done assuming given a subset of p's initial states (this is the connection to program specialization).

Each node in a process graph represents a *set of states* and is called a *configuration*. Any program graph has a single root node, called *initial configuration*, representing a subset of program p's initial states. A configuration c which branches to two or more configurations in a process graph represents a conditional transition from one set of program states to two or more sets of program states. An edge originating from a branching configuration c is labeled with a test on c. Abstractly this could be thought of as selecting the set of states causing control to follow this edge (a configuration which branches usually corresponds to a test in the program p).

Definition. A process graph g is *correct* for a program p with respect to an initial configuration ci iff
(i) the process graph is *complete*: if program p takes state s into s' in a computation originating from an initial state $si \in ci$, and if s is represented by configuration c, then there is a transition from c to c' in the process graph of p such that c' includes s'.
(ii) the nodes branched to from a branching configuration c are uniquely determined by the tests on c (the configuration c is divided into disjoint sets of states following the corresponding branches).

In general, a program p may have many different correct process graphs. A specific computation process follows a unique *walk* — a sequence of nodes and edges — in the process graph for program p. Because of the two requirements above, any computation process of p corresponds to exactly one walk in a correct process graph for program p (the requirement (i) says there is at least one walk, and (ii) says there is at most one walk). A process graph g for a program p is a *model* of p's computational behavior for a given set of initial states ci. In the following we refer to process graphs when we mean correct process graphs.

Graph Developers. How can one construct a process graph for a program and an initial configuration? Supercompilation [22,24] uses two methods for graph development: *driving* and *folding*.

Driving. This is a general method for constructing a (potentially infinite) *process tree* (a process graph that happens to be a tree) by step-wise exploring all possible computations of a program p starting from an initial configuration ci [20,22,24]. At any point during driving one has a perhaps incomplete process tree and a way to extend the process tree by adding a new node. Driving follows all possible computation processes starting from an initial configuration ci and continues until every leaf of the process tree represents only terminal states. Driving covers the activities of specializing and unfolding in partial evaluation.

Folding. The ultimate goal is to construct a finite process graph for a program p and an initial configuration ci. At any point during driving one may, instead of extending the process graph by driving, try to fold new transitions back into old graph configurations. Folding may include adding a new edge back from a non-terminal configuration N to another configuration M in the process graph, or merging two 'close enough' configurations N and M: given a configuration M with a path to N, one may replace the edges originating from M and create a new edge instead to a generalized configuration M' representing a superset of the states represented by M and N.

Given a finite process graph g that is correct for program p and the set of initial states ci, one may then construct a new program q from g (this is easy to achieve in practice). We will require that g will be correct for q, and q will be functionally equivalent to p with respect to the set of initial states ci. Our aim, of course, is to make q more efficient than the original program p.

3 Perfect Process Graph

How can one assess the 'quality' of a process graph? Clearly, if there is some edge in a process graph which is not used by any computation process, the process graph can't be considered as an 'optimal' model of the program's computational behavior. That is, the process graph contains at least one edge for which no initial state exists to follow it. We say that the more infeasible walks exist in a process tree, the worse is the process graph.

Definition. A walk w in a process graph g is *feasible* if at least one initial state exists which follows w.
Definition. A node n in a process graph g is *feasible* if it belongs at least to one feasible walk w in g.
Definition. A process graph is *perfect* if all its walks are feasible.

Infeasible walks not only increases the size of a process graph, but also reduces the efficiency of feasible walks. Consider the last feasible node in an otherwise infeasible walk. Since the node is feasible, at least one feasible walk goes through it. Since the infeasible walk goes through it as well, the node is a branching node: one branch is feasible, another is infeasible. Each branching has several conditions which have to be tested. This is extra work. Thus infeasible branches introduce additional tests and thereby degrade the efficiency of feasible walks. The more interpretive an algorithm is, the less perfect its process graph [23].

Example. Consider the following fragment of a graph (or the program represented by it — we will not distinguish here). The branches **'B** and **'C** are infeasible, and the tests **EQA?**$_2$ and **EQA?**$_3$ are redundant. There exists no initial state which follows the branches **'B** and **'C**.

```
(IF (EQA?1 x '5)
    (IF (EQA?2 x '5) 'A 'B)
    (IF (EQA?3 x '5) 'C 'D))
```

Perfect Graph Developers. How can one construct a perfect process graph for a program and a given initial configuration? Unfortunately, no algorithm exists that could transform any program p into an equivalent finite, perfect process graph (formally proven in [22]). That is, one can not build a *perfect graph developer*. However, *perfect tree developers* for 'well-formed' languages exist which develop perfect process trees for an arbitrary program p and an initial configuration ci. This is the case with the programming language presented in this paper. While the problem of perfect graph development cannot be solved in general, perfect tree development may be achieved. This is the main motivation for studying it.

Construction guideline. (1) Start by devising a perfect tree developer; (2) make the corresponding graph developer as 'perfect' as possible, without sacrificing computability and termination. Why do we consider this as essential? Because the first goal may be achieved constructively for 'well-formed' languages, while the second goal can not be achieved in general. Another aspect: one can not expect to make 'clever' folding decisions based on insufficient information obtained by driving. Once a perfect driving mechanism is constructed, it is a solid ground for the further refinement of a graph developer. As a result, the problem of approximation in the development of process graphs is driven into one corner: folding.

4 The Language S-Graph

The choice of the subject language is crucial for writing concise and clear algorithms for program specialization. In order to concentrate on the essence of driving, we limit ourselves to a pure symbol manipulation language, called S-Graph. As the name implies, one can think of S-Graph programs as being textual representations of graphs.

S-Graph is a first-order, functional programming language restricted to tail-recursion. The only data type are well-founded, i.e. non-circular, S-expressions (as known from Lisp). Despite its simplicity the language is complete and universal. The semantics of the language is straightforward. A program is a list of function definitions where each function body is an

```
Prog  ::=  [Def*]
Def   ::=  (DEFINE Fname [Var*] Tree)

Tree  ::=  (LET  Var  Exp  Tree)  |  (CALL Fname [Arg*])
           (IF   Cntr Tree Tree)  |  Exp

Cntr  ::=  (CONS? Arg Var Var)    |  (EQA? Arg Arg)

Exp   ::=  Arg  |  (CONS Exp Exp)
Arg   ::=  Val  |  Var

Val   ::=  (ATOM Atom)
Var   ::=  (VAR  Name)
```

Fig. 1. Syntax of flat S-Graph.

expression built from a few elements: conditionals **IF**, local bindings **LET**, function calls **CALL**, constructors **CONS** and atomic constants (drawn from an infinite set of symbols).

Note the conditional in S-Graph: the test **Cntr** may update the environment. As in supercompilation, we refer to such tests as *contractions* [24]. Two elementary contractions are sufficient for S-expressions:

- **(EQA? x y)** tests the equality of two atoms: **x**'s value and **y**'s value; if the arguments are non-atomic then the test is undefined.
- **(CONS? x h t)** if the value of **x** is a pair **(CONS a b)**, then the test succeeds and the variable **h** is bound to **a** and the variable **t** to **b**; otherwise, the test is false.

The arguments of function calls and contractions are restricted to variables and atomic constants in order to limit the number of places where values may be constructed. Because there are no nested function calls, we call this variant of the language *flat* (i.e. it corresponds to a flow-chart language). This is generalizable to nested function calls at the cost of more complex driving algorithms. In the following we will refer to the flat variant of the language simply as S-Graph.

Example. String pattern matching is a typical problem to which various specialization methods have been applied. The subject program is a naive pattern matcher which checks whether a string **p** (the pattern) occurs within another string **s**. The matcher is fairly simple: it returns **'SUCCESS** if **p** occurs in **s**, **'FAILURE** otherwise. The function **LOOP** compares the pattern with the beginning of the string. If the comparison fails the first element of the string is cut off and the function tries to match the remaining string against the pattern. This strategy is not optimal because the same elements in the string may be tested several times. In case of a mismatch the string is shifted by one and no further information is used for advancing in the string.

Syntactic sugar: we write **'Atom** as shorthand for **(ATOM Atom)**, and lowercase identifiers as shorthand for **(VAR Name)**.

```
(DEFINE MATCH [p s]
  (CALL LOOP [p s p s]))                    ; initialize loop

(DEFINE LOOP [p s pp ss]
  (IF (CONS? p phead ptail)
    (IF (CONS? s shead stail)
      (IF (EQA? phead shead)
        (CALL LOOP [ptail stail pp ss])     ; continue
        (CALL NEXT [pp ss]))                ; shift string
      'FAILURE)
    'SUCCESS))

(DEFINE NEXT [p s]
  (IF (CONS? s shead stail)
    (CALL LOOP [p stail p stail])           ; restart loop
    'FAILURE))
```

Fig. 2. Naive string matcher in S-Graph.

```
int  :: Tree → Env → Const
cntr :: Cntr → Env → Branch
data Branch  =  TRUE Env | FALSE Env

int (CALL f as) e    =  int t (mkEnv vs as e)
                        where (DEFINE _ vs t) = getDef f

int (LET v x t) e    =  int t (e&[v↦ x/e])

int (IF c t' t") e   =  case cntr c e of
                             TRUE  e' → int t' e'
                             FALSE e" → int t" e"
int x e              =  x/e

cntr (EQA? x y) e    =  case (x/e, y/e) of
                             (ATOM a, ATOM a) → TRUE  e
                             (ATOM _, ATOM _) → FALSE e

cntr (CONS? x h t) e =  case x/e of
                             CONS a b → TRUE (e&[h↦a, t↦b])
                             ATOM _   → FALSE e
```

Fig. 3. Interpretive definition of S-Graph.

Interpretive Definition. The semantics of S-Graph is defined by an interpretive definition (this will be the starting point for defining driving). In order to write the interpreter in a concise way we use some shorthand notations (the syntax is Haskell-like, the semantics is call-by-value):

- `[`v_1`↦`c_1`, … ,`v_n`↦`c_n`]` an environment consisting of a list of variables bindings,
- `e&[…]` the function `&` updates the environment `e` with the list of variable bindings `[…]`,
- `x/e` the function `/` substitutes all variables in the expression `x` by the values given by the bindings in the environment `e`.

The parameter containing the text of the interpreted program is omitted. The function `mkEnv` builds a new environment from a list of variables `vs`, a list of arguments `as` and an environment `e`; the function `getDef` returns the definition of a function given its name. The evaluation of an expression `Exp` does not 'compute' anything, it can only build up a structure. Note that ↦ is a sugared version of a constructor.

State. A computation state in S-Graph is fully defined by the current program point and the current environment. Variables originating from a program are called *program variables* (p-variables) and are bound to constants in the environment. Since we consider only tail-recursive S-Graph programs, states include only one program point `PPoint` (such as `IF`, `CALL`, `LET`).

```
State  ::=  PPoint Env
Env    ::=  [Bind*]

Bind   ::=  Var ↦ Const
Const  ::=  ATOM Atom  |  CONS Const Const
```

Fig. 4. State in S-Graph.

5 Information Propagation

The main hindrance in removing redundant tests is the lack of sufficient information about unknown values during program specialization. Starting from constant propagation we will develop step-by-step a driving mechanism for S-Graph which ensures the full propagation of information along the specialized program branches. Each step represents a different degree of information propagation.

5.1 Constant Propagation

Constants are the most elementary form of information that can be propagated during program specialization. During specialization we do not deal with precise states, but with configurations representing sets of states. If we do not want to define perfect driving, then we may approximate the sets of states using covering configurations that represent larger sets of states.

Configuration. A simple method for representing sets of states constructively uses *expressions with free variables* [24]. We introduce placeholders, called *configuration variables* (c-variables), which range over arbitrary constants. A p-variable in the environment of a configuration may be bound to a constant or to a c-variable (representing a 'dynamic' values). This is sufficient for constant propagation.

```
Conf   ::=  PPoint Cenv
Cenv   ::=  [Bind*]

Bind   ::=  Var ↦ Cval
Cval   ::=  Const       |  CVAR Name
Const  ::=  ATOM Atom   |  CONS Const Const
```

Fig. 5. Configuration for constant propagation.

Driving. The first version is obtained by extending the S-Graph interpreter (Fig. 3) to propagate constants wherever possible and to produce residual code where the involved constants are 'dynamic' (Fig. 7).

- If the result of evaluating an expression in a **LET** is not a constant then the p-variable is bound to a fresh c-variable (generated by the function **newcvar**).
- If a contraction (**EQA?**, **CONS?**) can not be decided then both branches have to be driven (function **cntr** returns **BOTH** and an environment for each branch).

This implements what is known as *constant propagation*, and corresponds to first-order partial evaluators based on constant propagation (e.g. [12]).

Remark. It was noticed [12] that the test **const?** does not require the values proper and may be approximated in a separate pre-processing phase, called *binding-time analysis*. This granted the first self-application of a partial evaluator.

5.2 Partially Static Structures

A simple extension is the propagation of partially static structures in driving. This corresponds to first-order partial evaluators using partially static structures (e.g. [16,10,2,4]). This extension completes the construction of the function **dev** (in the following we will refine the handling of contractions during driving).

Configuration. The description of a configuration is refined by replacing the definition of **Cval**.

```
Conf   ::=  PPoint Cenv
Cenv   ::=  [Bind*]

Bind   ::=  Var ↦ Cval
Cval   ::=  ATOM Atom   |  CVAR Name   |  CONS Cval Cval
```

Fig. 6. Configuration for partially static structures.

Driving. The propagation of partially static structures is obtained by replacing the **LET** clause in function **dev** (Fig. 7) by

```
dev (LET v x t) e  =  dev t (e&[v ↦ x/e])
```

```
dev   :: Tree → Cenv → Tree
cntr  :: Cntr → Cenv → Branch
const :: Cval → Bool
data Branch  =  TRUE Cenv | FALSE Cenv | BOTH Cntr Cenv Cenv

dev (CALL f as) e  =  dev t (mkEnv vs as e)
                      where (DEFINE _ vs t) = getDef f

dev (LET v x t) e  =  let x' = x/e in
                      if const? x'
                         then dev t (e&[v ↦ x'])
                         else LET v' x' (dev t (e&[v ↦ v']))
                              where v' = newcvar

dev (IF c t' t") e =  case cntr c e of
                           TRUE  e'        → dev t' e'
                           FALSE e"        → dev t" e"
                           BOTH  c' e' e" → IF  c'  (dev t' e')
                                                    (dev t" e")
dev x e            =  x/e

cntr (EQA? x y) e  =
     let x' = x/e; y' = y/e in
     case (x', y') of
          (ATOM a, ATOM a) → TRUE  e
          (ATOM _, ATOM _) → FALSE e
          (CVAR _, CVAR _) → BOTH (EQA? x' y') e e
          (CVAR _, ATOM _) → BOTH (EQA? x' y') e e
          (ATOM _, CVAR _) → BOTH (EQA? x' y') e e

cntr (CONS? x h t) e =
     let x' = x/e in
     case x' of
          CONS a b → TRUE (e&[h ↦ a, t ↦ b])
          ATOM _   → FALSE e
          CVAR _   → BOTH (CONS? x' h' t') e' e
                     where h' = newcvar; t' = newcvar
                           e' = e&[h ↦ h', t ↦ t']
const? (ATOM _)    =  True
const? (CVAR _)    =  False
const? (CONS a b)  =  and (const? a) (const? b)
```

Fig. 7. Constant propagation in S-Graph.

5.3 Propagation of Contraction Information

Using constant propagation and partially static structures we are able to prune many infeasible branches, but not all (see example in Sect. 3).

Configuration. In addition to the propagation of information by substitution (which we refer to as *positive* information, or *assertions*), we need to propagate the negation of this information (*restrictions*). We refine configurations by adding a list of restrictions on c-variables. A restriction of the form `Rval # Rval` states which values must not be equal. The restriction list may contain zero, one or more restrictions for each c-variable. Otherwise the configuration remains unchanged (Fig. 8). The ↦ and `#` are sugared versions of constructors.

Assertions. Propagating assertions requires updating the bindings of p-variables (corresponding to the well-known concept of unification). To capture the information that two unknown values are equal, we exploit the *equality of c-variables*. This is done by adding an extra case for equal c-variables to the `EQA?` clause. This goes beyond constant propagation. For example, the assertion `x='5` is passed into the then-branch simply by replacing the c-variable `cx` by `'5`:

`([x↦ cx],[])  (EQA? x '5)` ⇒ *then-branch:* `([x↦ '5],[])`

```
Conf   ::=  PPoint Cenv
Cenv   ::=  [Bind*] [Restr*]

Bind   ::=  Var ↦ Cval
Cval   ::=  ATOM Atom  |  CVAR Name  |  CONS Cval Cval

Restr  ::=  Rval # Rval
Rval   ::=  ATOM Atom  |  CVAR Name  |  CONS
```

Fig. 8. Configuration for perfect driving.

Restrictions. Propagating restrictions requires updating a list of restrictions on c-variables. For example, the restriction `x≠'5` is passed into the else-branch by adding a restriction on the c-variable `cx`:

`([x↦cx],[])  (EQA? x '5)` ⇒ *else-branch :* `([x↦cx],[cx # '5])`

The mechanism for checking restrictions is separated into the function `both` which is common for both contractions. If a contraction can not be decided using the list of p-variable bindings, then the list of restrictions is checked whether, possibly, a restriction exists which can be used to decide the contraction. In case such a restriction is found one can cut off the infeasible then-branch. This is done in function `both` by checking whether a substitution in the list of restrictions leads to a contradiction.

Auxiliary functions. To simplify the definition we provide the following three functions for manipulating a configuration environment `e`: the function `&` adds new bindings to `e`, the function `\` adds new restrictions to `e`, and the substitution `/` is extended to substitute variables in the configuration environment. In our case these functions may be defined as follows (the function `++` is list append, the function `b2r` converts a `Bind` into a `Restr`):

```
(b,r) & bs  =  ((b++bs), r)
(b,r) / bs  =  ((b/bs), (r/(b2r bs)))
(b,r) \ bs  =  ( b,     (r++(b2r bs)))
```

During driving, c-variables are generated and may disappear as result of a substitution, leaving 'dangling' restrictions or tautologies, such as `'A # 'B`. They may be cleared out (e.g. after `/`), though this does not interfere with driving.

Correctness and Perfectness. In order to verify S-Graph driving we have to prove that the mechanism is correct and perfect. The correctness of driving with respect to the interpretive definition of S-Graph ensures that the process tree contain at least all necessary (and maybe some infeasible) branches. The perfectness of driving guarantees that the process tree contains no infeasible branches. The existence of a perfectness theorem guarantees that driving propagates all information sufficient for pruning the process tree to its minimal size (omitted due to lack of space). This completes the task of defining perfect driving for S-Graph (Fig. 9).

Remark. In order to keep the presented perfect tree developer as simple as possible and at the same time to preserve the termination properties of the subject programs in the residual programs, we require that subject programs do not go 'wrong', i.e. atomic equality `EQA?` is not applied to non-atomic arguments. This may be guaranteed by adding a `CONS?` test for each non-atomic argument of `EQA?` in the subject programs.

6 Perfect Driving of a Naive Pattern Matcher

By specializing the naive pattern matcher (Fig. 2) with respect to a fixed pattern we show that perfect driving coupled with a simple folding strategy obtains the efficiency of a matcher generated by the Knuth, Morris & Pratt (KMP) algorithm [15]. This effect is achieved without the need for an 'insightful reprogramming' of the naive matcher as necessary for partial evaluation [5,11]. The complexity of the specialized algorithm is $O(n)$, where n is the length of the string. The naive algorithm has complexity $O(m*n)$, where m is the length of the pattern.

```
dev  :: Tree → Cenv → Tree
cntr :: Cntr → Cenv → Branch
both :: Cntr → Cenv → Cenv → [Bind] → Branch
contradict :: Cenv → Bool
data Branch  =  TRUE Cenv | FALSE Cenv | BOTH Cntr Cenv Cenv

dev (CALL f as) e  =  dev t (mkEnv vs as e)
                      where (DEFINE _ vs t) = getDef f

dev (LET v x t) e  =  dev t (e&[v ↦ x/e])

dev (IF c t' t") e =  case cntr c e of
                           TRUE  e'        → dev t' e'
                           FALSE e"        → dev t" e"
                           BOTH  c' e' e" → IF  c'  (dev t' e')
                                                    (dev t" e")
dev x e            =  x/e

cntr (EQA? x y) e  =
     let x' = x/e; y' = y/e in
     case (x', y') of
          (ATOM a, ATOM a) → TRUE  e
          (ATOM _, ATOM _) → FALSE e
          (CVAR a, CVAR a) → TRUE  e
          (CVAR _, CVAR _) → both (EQA? x' y') e e [x' ↦ y']
          (CVAR _, ATOM _) → both (EQA? x' y') e e [x' ↦ y']
          (ATOM _, CVAR _) → both (EQA? x' y') e e [y' ↦ x']

cntr (CONS? x h t) e  =
     let x' = x/e in
     case x' of
          CONS a b → TRUE (e&[h ↦ a, t ↦ b])
          ATOM _   → FALSE e
          CVAR _   → both (CONS? x' h' t') e' e [x' ↦ CONS h' t']
                     where h' = newcvar; t' = newcvar
                           e' = e&[h ↦ h', t ↦ t']
both c' te fe b  =
     let e' = te/b; e" = fe\b in
     if  contradict e' then FALSE e"
                       else BOTH  c' e' e"
contradict (b,r) =
     or (map contradict' r)
     where contradict' (x # x)  =  True
           contradict' (_ # _)  =  False
```

Fig. 9. Perfect driving in S-Graph.

Folding. At any point during driving one has a way to examine a non-terminal configuration and to decide to do one of: (i) fold the current configuration into an existing configuration; (ii) drive the configuration further. Two questions are relevant for folding:

1) Which *program points* do we consider for folding?
2) What is the *criterion* for folding?

It is sufficient to couple perfect driving with *folding of identical configurations* for obtaining efficient matchers from a naive pattern matcher and a given pattern. Driving can be coupled with more sophisticated folding strategies, but this is beyond the scope of this paper (and not needed for the example). Folding of identical configurations answers the questions as follows:

1) Dynamic conditionals are considered for folding.
2) Two configurations represent the same set of states.

The method of dynamic conditionals is a well-known technique in program specialization [22,3]: only those program points are considered for folding which introduce a branching (the conditional can not be decided, it is 'dynamic').

When the descriptions of two configurations are identical (i.e. contain the same bindings and restrictions, modulo variable renaming) they represent the same set of states and one may, instead of continuing driving, add a new transition back from the current configuration into the old configuration.

```
(DEFINE F1 [s]
  (IF (CONS? s shead-1 stail-2)
    (IF (EQA? shead-1 'A)
      (IF (CONS? stail-2 shead-3 stail-4)
        (IF (EQA? shead-3 'A)
          (CALL F5 [stail-4])
          (CALL F1 [stail-4]))
        'FAILURE)
      (CALL F1 [stail-2]))
    'FAILURE))

(DEFINE F5 [stail-4]
  (IF (CONS? stail-4 shead-5 stail-6)
    (IF (EQA? shead-5 'B)
      'SUCCESS
      (IF (EQA? shead-5 'A)
        (CALL F5 [stail-6])
        (CALL F1 [stail-6])))
    'FAILURE))
```

Fig. 10. KMP-like residual program for the pattern **AAB**.

7 Related Work

The principles of driving were first formulated in the early seventies by Turchin [20,21] and further developed in the eighties [22,24]. From its very inception, supercompilation has been tied to a specific programming language, called Refal [24]. Applications of supercompilation include, among others, program specialization, program inversion and theorem proving. Other related aspects have been investigated in [1,7,8,13,14,17,18,26]. The notion of perfect process graphs and perfect driving were introduced in [22,23].

The language S-Graph is closely related to Turchin's Refal graphs [25]. But due to S-Graph's simpler data structure, untyped variables and only two elementary contractions, one may build rather clear and concise driving algorithms. In particular, there is only one way to compose and decompose S-expressions (as opposed to Refal data structures). There is a close relation between driving and the neighborhood analysis for S-Graph [1]. Another 'graph-like' language representing decision trees, was used by Bondorf for the implementation of a self-applicable partial evaluator Treemix [2].

Specializing a naive string matcher is a typical problem to which various methods of program manipulation have been applied. A partial evaluator can achieve the same non-trivial result after identifying static components in the naive matcher and reprogramming the subject program [5,11]. Clearly, doing this by using an "automatic insight" frees the user from performing such subtle tasks. Generalized Partial Computation (GPC), another principle for program specialization based on partial evaluation and theorem proving, achieves the same optimal version [6]. In its essence GPC is related to driving, but differs in the propagation of arbitrary predicates, assuming the use of a theorem prover. Disunification in GPC was considered in [19]. It is not surprising, that the same optimal pattern matcher can be achieved by a Refal supercompiler [9]. Note that we used only a small part of the supercompilation methodology: perfect driving for a language with tree-structured data coupled with a simple folding strategy.

8 Conclusion

We introduced a simple model for assessing the 'quality' of program specialization: the closer a process tree is to a perfect one, the better the method. This enables us to rate various specialization techniques. Although specialization methods vary from language to language, they all have the same goal in common: propagating as much information as possible in order to increase the efficiency of the resulting programs.

We showed that a mechanism for perfect driving can be constructed for a simple language, called S-Graph. On the one hand the propagation of additional information requires extra work during specialization, but on the other hand less time is spent for developing infeasible branches. Most important, the efficiency of the resulting programs may be improved considerably. In particular, perfect driving coupled with a simple folding strategy obtains the efficiency of a matcher generated by the Knuth, Morris & Pratt algorithm without 'insightful reprogrammig' of the naive matcher. This reveals that the power of Turchin's supercompilation method is independent of the language used to express it (i.e. Refal) and that the principles may be applied to other languages. Partial evaluation and supercompilation do not contradict each other and the question of integrating them is on the agenda. How far these principles can be taken, how they can be applied to more realistic languages and what their limitations are will be a task for future research.

Acknowledgments. This work could not have been carried out without the pioneering work of Valentin Turchin and we are very grateful for many stimulating discussions. It is a great pleasure to thank Neil Jones for thorough comments and for clarifying an earlier version of this paper. We greatly appreciate fruitful discussions with the members of the Refal group in Moscow and the TOPPS group at DIKU. Many thanks are due to Sergei Abramov, Anders Bondorf, Ruten Gurin, Jesper Jørgensen, Victor Kistlerov, Arkady Klimov, Alexander Romanenko, Sergei Romanenko and David Sands. Special thanks to Thomas Eisenzopf for implementing the graph developers.

References

1. Abramov S. M., Metacomputation and program testing. In: *1st International Workshop on Automated and Algorithmic Debugging*. (Linköping, Sweden). 121-135, Linköping University 1993.
2. Bondorf A., A self-applicable partial evaluator for term rewriting systems. In: Díaz J., Orejas F. (ed.), *TAPSOFT '89*. (Barcelona, Spain). Lecture Notes in Computer Science, Vol. 352, 81-95, Springer-Verlag 1989.
3. Bondorf A., Danvy O., Automatic autoprojection of recursive equations with global variables and abstract data types. In: *Science of Computer Programming*, 16(2): 151-195, 1991.
4. Consel C., Binding time analysis for higher order untyped functional languages. In: *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. (Nice, France). 264-272, ACM Press 1990.
5. Consel C., Danvy O., Partial evaluation of pattern matching in strings. In: *Information Processing Letters*, 30(2): 79-86, 1989.
6. Futamura Y., Nogi K., Takano A., Essence of generalized partial computation. In: *Theoretical Computer Science*, 90(1): 61-79, 1991.
7. Glück R., Towards multiple self-application. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. (New Haven, Connecticut). 309-320, ACM Press 1991.
8. Glück R., Projections for knowledge based systems. In: Trappl R. (ed.), *Cybernetics and Systems Research '92*. Vol. 1, 535-542, World Scientific: Singapore 1992.
9. Glück R., Turchin V. F., Application of metasystem transition to function inversion and transformation. In: *Proceedings of the ISSAC '90*. (Tokyo, Japan). 286-287, ACM Press 1990.

10. Jones N. D., Automatic program specialization: a re-examination from basic principles. In: Bjørner D., Ershov A. P., Jones N. D. (ed.), *Partial Evaluation and Mixed Computation*. (Gammel Avernæs, Denmark). 225-282, North-Holland 1988.
11. Jones N. D., Gomard C. K., Sestoft P., *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice Hall: New York, London, Toronto 1993.
12. Jones N. D., Sestoft P., Søndergaard H., Mix: a self-applicable partial evaluator for experiments in compiler generation. In: *Lisp and Symbolic Computation*, 2(1): 9-50, 1989.
13. Klimov And. V., Dynamic specialization in extended functional language with monotone objects. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. (Yale University, Connecticut). 199-210, ACM Press 1991.
14. Klimov And. V., Romanenko S. A., 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).
15. Knuth D. E., Morris J. H., Pratt V. R., Fast pattern matching in strings. In: *SIAM Journal on Computing*, 6(2): 323-350, 1977.
16. Mogensen T. Æ., Partially static structures in a self-applicable partial evaluator. In: Bjørner D., Ershov A. P., Jones N. D. (ed.), *Partial Evaluation and Mixed Computation*. (Gammel Avernæs, Denmark). 325-347, North-Holland 1988.
17. Romanenko A. Yu., Inversion and metacomputation. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. (Yale University, Connecticut). 12-22, ACM Press 1991.
18. Romanenko S. A., Driving for Refal-4 programs. Keldysh Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow. Preprint No. 211, 1987 (in Russian).
19. Takano A., Generalized partial computation using disunification to solve constraints. In: Rusinowitch M., Rémy J. L. (ed.), *Conditional Term Rewriting Systems. Proceedings*. (Pont-à-Mousson, France). Lecture Notes in Computer Science, Vol. 656, 424-428, Springer-Verlag 1993.
20. Turchin V. F., Equivalent transformations of recursive functions defined in Refal. In: *Teoria Jazykov i Metody Programmirovanija (Proceedings of the Symposium on the Theory of Languages and Programming Methods)*. (Kiev-Alushta, USSR). 31-42, 1972 (in Russian).
21. Turchin V. F., Equivalent transformations of Refal programs. In: *Avtomatizirovannaja Sistema upravlenija stroitel'stvom. Trudy CNIPIASS*, 6: 36-68, 1974 (in Russian).
22. Turchin V. F., The language Refal, the theory of compilation and metasystem analysis. Courant Institute of Mathematical Sciences, New York University. Courant Computer Science Report No. 20, 1980.
23. Turchin V. F., Semantic definitions in Refal and automatic production of compilers. In: Jones N. D. (ed.), *Semantics-Directed Compiler Generation*. (Aarhus, Denmark). Lecture Notes in Computer Science, Vol. 94, 441-474, Springer-Verlag 1980.
24. Turchin V. F., The concept of a supercompiler. In: *ACM TOPLAS*, 8(3): 292-325, 1986.
25. Turchin V. F., The algorithm of generalization in the supercompiler. In: Bjørner D., Ershov A. P., Jones N. D. (ed.), *Partial Evaluation and Mixed Computation*. (Gl. Avernæs, Denmark). 341-353, 1988.
26. Turchin V. F., Program transformation with metasystem transitions. In: *Journal of Functional Programming*, 3(3): 283-313, 1993.