Partial Deduction and Driving are Equivalent

Robert Glück* & Morten Heine Sørensen

DIKU, Department of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
E-mail: {glueck,rambo}@diku.dk

Abstract. Partial deduction and driving are two methods used for program specialization in logic and functional languages, respectively. We argue that both techniques achieve essentially the same transformational effect by unification-based information propagation. We show their equivalence by analyzing the definition and construction principles underlying partial deduction and driving, and by giving a translation from a functional language to a definite logic language preserving certain properties. We discuss residual program generation, termination issues, and related other techniques developed for program specialization in logic and functional languages.

1   Introduction

Partial evaluation is a method of program specialization which has received much attention in the area of functional languages [Jon93]. Partial evaluation in the context of functional languages usually relies on constant propagation, while transformation techniques for logic languages exploit unification-based information propagation.

The relation of specialization methods in functional languages has been studied to some extent, e.g. [Hol91, Glu93, Sor94c], but few attempts have been made to study the relationship of techniques used in logic and functional languages. Our goal was to establish a correspondence between two powerful methods used in the two programming paradigms, namely *driving* as used in *supercompilation* and *partial deduction*. Although it has been observed that supercompilation is capable of theorem proving and program inversion [Tur72, Tur80b, Tur82], an in-depth comparison was not done. Such a comparison is useful, because it not only prevents reinventing the wheel, but also possibly generates new insights and developments, e.g. with respect to termination and generalization (abstraction) which are current research topics in both worlds.

In the following we refer to Lloyd-Shepherdson's partial deduction [Llo91, Kom92] and to positive driving [Glu93, Sor94c], a variant of Turchin's driving [Tur86] for a functional language with lists. Both transformation techniques developed independently, at different places and times. Komorowski introduced

partial deduction into logic programming in 1981 [Kom81]. After several years
of neglect the importance was realized and there is now growing interest in this
area, e.g. [Kom92]. The main theoretical results for partial deduction were de-
veloped by Lloyd and Shepherdson [Llo91].

Driving was conceived in the early seventies by Turchin in the former USSR
[Tur72] and developed into a comprehensive methodology summarized as super-
compilation [Tur80a, Tur86]. It strictly contains partial evaluation as well as
Wadler's more recent invention *deforestation* [Wad90], but driving and super-
compilation have taken longer to be recognized in the context of program spe-
cialization, e.g. [Jon94]. However, the functional language used in this paper does
not allow function calls inside constructor expression, and hence deforestation
as known in functional languages is not applicable. We chose this simplification
of the functional language since logic languages do not allow predicates inside
constructor expressions either.

The remainder of the paper is organized as follows. First we define the func-
tional language $\mathcal{M}$, followed by a development of driving for this language. Then
we review logic programming and partial deduction. We then describe the corre-
spondence between the two methods, and give applications of the correspondence
including a discussion of termination problems in the two worlds and an extended
example of logic programming by driving.

2   A simple functional language $\mathcal{M}$

In this section we describe a fragment of a first-order functional language often
studied in connection with deforestation, supercompilation, and partial evalua-
tion. We first describe the syntax of the language, then some notational conven-
tions, and finally a simple rewrite semantics.

Definition 1 (Language $\mathcal{M}$). Let t, b, p, d range over $\mathcal{M}$ terms, constructor
terms, patterns, and definitions, respectively:

$$
\begin{aligned}
d ::= &\ f\ v_1 \ldots v_n \leftarrow t \\
      &|\ \ g\ p_1\ v_1 \ldots v_n \leftarrow t_1 \\
      &\vdots \\
      &\ \ g\ p_m\ v_1 \ldots v_n \leftarrow t_m
\end{aligned}
\qquad
\begin{aligned}
t &::= b \mid f\ b_1 \ldots b_n \mid g\ t\ b_1 \ldots b_n \\
b &::= v \mid c\ b_1 \ldots b_n \\
p &::= c\ v_1 \ldots v_k
\end{aligned}
$$

A term with no variables is called *ground*. As is customary, we require that
patterns be *linear*, *i.e.* that no variable occurs in a pattern more than once. We
also require that all variables of a right hand side be present in the corresponding
left hand side. To ensure uniqueness of reduction we require from a program that
each f-function has at most one definition, and in the case of a g-function that
no two patterns p_i and p_j contain the same constructor.

There are a number of noteworthy restrictions on this language. First, func-
tion definitions may have at most one argument defined on patterns, and only
with non-nested patterns. This restriction is quite common; methods exist for

translating multiple, nested patterns to this form. Second, only linear patterns are allowed. One can extend the techniques from the present paper to allow non-linear patterns or, equivalently, incorporate a conditional construct which can test terms for equality, see [Glu93, Sor94c]. Finally, function calls cannot occur as arguments in other function calls except as the first argument to a g-function call, and function calls cannot occur inside constructors. For instance, a term can have form $g\ (f\ b'_1 \ldots b'_{n'})\ b_1 \ldots b_n$ but the b's must be constructor terms, they cannot contain function calls. This means that programs and terms are *order-of-evaluation independent*, and that there is a complete separation of data-flow and control-flow. These properties make it easy to translate $\mathcal{M}$ terms and programs into logic goals and programs. It is straight-forward to extend the principles of (positive) supercompilation to account for nested calls, see [Sor94c].

In conclusion, the language is simple, but the correspondences with which we are concerned carry over to larger languages, at the expense of some technical complications.

Definition 2 (Notational conventions). If a program s contains an f-function definition

$$f\ v_1 \ldots v_n \leftarrow t$$

then in the context of s, t^f denotes t and $v_1^f \ldots v_n^f$ denote $v_1 \ldots v_n$. Similarly, if a program s contains a g-function definition

$$g\ p_1\ v_1^1 \ldots v_n^1 \quad \leftarrow\ t_1$$
$$\vdots$$
$$g\ p_m\ v_1^m \ldots v_n^m \leftarrow t_m$$

where $p_i\ =\ c_i\ v_{n+1}^i \ldots v_{n+k_i}^i$, then in the context of s, t^{g,c_i} denotes t_i, and $v_1^{g,c_i} \ldots v_n^{g,c_i}$ denote $v_1^i \ldots v_n^i$. Finally, $v_{n+1}^{g,c_i} \ldots v_{n+k_i}^{g,c_i}$ denote $v_{n+1}^i \ldots v_{n+k_i}^i$, and $p_1^g \ldots p_m^g$ denote $p_1 \ldots p_m$.

Although $\mathcal{M}$ includes general data structures we are most often concerned with *lists*. Therefore it is convenient to introduce the infix notation $x : xs$ (associating to the right) as an abbreviation for $Cons\ x\ xs$, $[]$ as an abbreviation for Nil, and $[x_1, \ldots, x_n]$ as an abbreviation for $x_1 : \ldots x_n : []$.

The expression $t\{v_i := t_i\}_{i=1}^n$ denotes the substitution of all occurences of variable v_i in the term t by the corresponding term t_i given in the list of bindings. The notation $\{l_1, \ldots, l_m\}$ denotes ordered lists, and we let $\{l_j\}_{j=1}^m = \{l_1, \ldots l_m\}$.

We now state the semantics of $\mathcal{M}$ in terms of a rewrite interpreter. For every ground term two possibilities exist during evaluation. Either the term is a ground constructor term (a *value*), and then interpretation stops, or the term has an innermost call which is the next to be unfolded (this call is the one chosen by call-by-name as well as call-by-value evaluation). In the second case the term will be written $e[r]$ where r identifies the next function call to unfold, and e is the surrounding part of the term. Traditionally, these are called the *redex* and the *evaluation context*, respectively. The intention is that r is a call which is ready to be unfolded (no further evaluation of the arguments is necessary). We now make these notions precise.

Definition 3 (Context,redex,value). Let e, r, a range over contexts, redexes, and values, respectively:

$$e ::= [] \mid g\ e\ a_1 \ldots a_n$$
$$r ::= f\ a_1 \ldots a_n \mid g\ a_0\ a_1 \ldots a_n$$
$$a ::= c\ a_1 \ldots a_n$$

The result of substituting t for $[]$ in a context is denoted $e[t]$.

Proposition 4. *For a ground term t, either there exists exactly one context and redex e, r such that $t \equiv e[r]$, or $t \equiv a$.*

We now introduce *interpretation sequences* (I-sequences), which trace the evaluation of a ground term in some program, and *results* of evaluation.

Example 1. As an example consider the *last* function.

$$last\ (x : xs) \leftarrow l\ xs\ x$$
$$l\ []\ x \qquad\quad \leftarrow x$$
$$l\ (z : zs)\ x \quad \leftarrow l\ zs\ z$$

The result of the term $last\ [A, B]$ is B, which is the last term of the I-sequence:

$$last\ [A, B], \qquad l\ [B]\ A, \qquad l\ []\ B, \qquad B$$

Definition 5 (I-sequence). Let t_0 be a ground term. Define the *I-sequence* $\mathcal{I}[\![\ t_0\ ]\!]$ for t_0 inductively as follows. The first element is t_0. If an element has form $e[r]$ then the element has successor $\mathcal{N}[\![\ e[r]\ ]\!]$ where

$$\mathcal{N}[\![\ e[f\ a_1 \ldots a_n]\ ]\!] = e[t^f\{v_i^f := a_i\}_{i=1}^n]$$
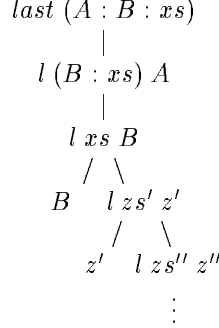$$\mathcal{N}[\![\ e[g\ (c\ a_{n+1} \ldots a_{n+k})\ a_1 \ldots a_n]\ ]\!] = e[t^{g,c}\{v_i^{g,c} := a_i\}_{i=1}^{n+k}]$$

If the I-sequence for some ground term t_0 is finite, we let $\mathcal{R}[\![\ t_0\ ]\!]$ denote the last term of the sequence (this must be a value) and call this the *result* of t_0.

3 Supercompilation

Supercompilation consists of three core constituents: *driving*, *folding*, and *generation of residual programs*. The driving component takes a program and a term and constructs a certain tree. Extended with the folding component, certain graphs are constructed instead. The last component generates a residual program from such a graph, provided that it is finite (in some expositions no intermediate graph is constructed). Apart from these three components, supercompilation usually consists of various other components: *postunfolding* and *generalization*.

We first describe driving and folding. A *transformation tree* (*T-tree*) for a term t (possibly containing variables) and a program s is a model of all possible computations with t in s.

Example 2. For the term *last* $(A : B : xs)$ and the *last* function from Example 1 we have the following T-tree:

$$last \; (A : B : xs)$$
$$|$$
$$l \; (B : xs) \; A$$
$$|$$
$$l \; xs \; B$$
$$/ \;\; \backslash$$
$$B \quad l \; zs' \; z'$$
$$/ \;\; \backslash$$
$$z' \quad l \; zs'' \; z''$$
$$\vdots$$

We shall be concerned not only with trees but also with graphs. A rooted *transformation graph* (*T-graph*) is the same as a T-tree except that one may, instead of developing a node further, try to fold new transitions back into existing nodes. Here we shall be concerned only with *identical folding*: when the same term modulo variable renaming is encountered twice in a branch (path from the root), then we make an arc from the parent of the second occurrence to the first occurrence and do not develop the branch from the parent of the second occurrence any further. This is just one possible scheme, but it often occurs in the literature (especially in connection with deforestation) [Wad90, Glu93, Chi93, Sor93, Sor94c].

Example 3. In the above example, the T-graph is obtained from the tree as follows. Call the node just above the vertical dots N. Then the parent of N gets an arc to itself, and the subtree with N as a root is deleted.

We now make these notions precise. The notions below extend those of Section 2 by the presence of free variables.

Definition 6. Let e, r, b range over contexts, redexes, and constructor terms, respectively:

$$e ::= [] \mid g \; e \; b_1 \ldots b_n$$
$$r ::= f \; b_1 \ldots b_n \mid g \; b_0 \; b_1 \ldots b_n$$
$$b ::= v \mid c \; b_1 \ldots b_n$$

The result of substituting t for $[]$ in a context is denoted $e[t]$.

Proposition 7. *For a term t, either there exists exactly one context and redex e, r such that $t \equiv e[r]$, or $t \equiv b$.*

Definition 8 (T-tree and T-graph). Let t_0 be a term. Define the *T-tree* $\mathcal{T}[\![\; t_0 \;]\!]$ inductively as follows. The root contains t_0. If a node N contains $t \equiv e[r]$ and $\mathcal{N}[\![\; t \;]\!] = \{t_1, \ldots, t_n\}$, then N has left to right children $N_1 \ldots N_n$, where N_i contains t_i, and where

$$\mathcal{N}[\![\, e[f\; b_1 \dots b_n]\,]\!] \qquad\qquad = \{e[t^f\{v_i^f := b_i\}_{i=1}^n]\}$$

$$\mathcal{N}[\![\, e[g\; (c\; t_{n+1}\dots t_{n+k})\; b_1\dots b_n]\,]\!] = \{e[t^{g,c}\{v_i^{g,c} := b_i\}_{i=1}^{n+k}]\}$$

$$\mathcal{N}[\![\, e[g\; v\; b_1\dots b_n]\,]\!] \qquad\qquad = \{(e[\hat{t}^{g,c_j}\{v_i^{g,c_j} := b_i\}_{i=1}^n])\{v := \hat{p}_j^g\}\}_{j=1}^m$$

Here $\hat{p}_j^g$ is obtained as a renaming θ of p_j^g such that $\hat{p}_j^g$ has no free variables in common with $e[g\; v\; b_1\dots b_n]$, and $\hat{t}^{g,c_j}$ is defined as $t^{g,c_j}\theta$.[2]

The *T-graph* $\mathcal{G}[\![\, t_0\,]\!]$ is defined as the graph obtained from $\mathcal{T}[\![\, t_0\,]\!]$ as follows. Start from the root and traverse all branches. Whenever a node N_2 is reached which contains a term t, where $t \equiv e[r]$ and a term identical to t modulo variable renaming is contained in a previous node N_1 in the same branch, then let the parent of N_2 have an arc back to N_1, and delete the subtree with root N_2.

T-trees can be perceived as a generalization of I-sequences to take terms with free variables into account. On ground terms, they are the same:

Proposition 9. *For any ground term t_0, $\mathcal{I}[\![\, t_0\,]\!] = \mathcal{T}[\![\, t_0\,]\!]$.*

Example 4. The T-tree for *last* $[A, B]$ is the same as the I-sequence.

The operation of computing T-trees is called *driving*. The description of T-trees and T-graphs here follows the description of *process trees* and *graphs*, respectively, as described in [Glu93], which in turn stem from *graphs of states* in Turchin's papers [Tur80a]. See also [Sor94b].

We now show how one can derive a residual term t' and a residual program p' from a finite T-graph. The basic idea is that every term $e[r]$ in the T-graph, where r is a f- or g-function call, gives rise to a new function definition, and the body, or bodies, of the new definition can be derived from the immediate descendants of $e[r]$ in the graph.

Definition 10 (Residual program from finite T-graph). Let t be the term in the root of a transformation graph $G = \mathcal{G}[\![\, t_0\,]\!]$. Then the residual term is $\mathcal{C}[\![\, t\,]\!]$, and the residual program contains:

(i) for every node N containing $e[f\; t_1 \dots t_n]$ with descendant containing t':

$$f^\square\; u_1 \dots u_l \leftarrow \mathcal{C}[\![\, t'\,]\!]$$

(ii) for every node N containing $e[g\; (c\; t_{n+1}\dots t_{n+k})\; t_1\dots t_n]$ with descendant containing t':

$$f^\square\; u_1 \dots u_l \leftarrow \mathcal{C}[\![\, t'\,]\!]$$

---

[2] We assume that given a term t and pattern p, we have a procedure for choosing a *unique* renaming $\hat{p}$ with with no free variables in common with t.

(iii) for every node N containing $e[g\ v\ t_1 \ldots t_n]$ with descendants containing $t'_1 \ldots t'_m$:

$$g^{\square}\ \hat{p}^g_1\ u_1 \ldots u_l\ \leftarrow \mathcal{C}[\![\ t'_1\ ]\!]$$
$$\vdots$$
$$g^{\square}\ \hat{p}^g_m\ u_1 \ldots u_l\ \leftarrow \mathcal{C}[\![\ t'_m\ ]\!]$$

where $f^{\square}, g^{\square}$ are fresh function names, $u_1 \ldots u_l$ are the free variables of the term in N (except v in case *(iii)*), and $\mathcal{C}$ is defined as follows.

$$
\begin{aligned}
\mathcal{C}[\![\ b\ ]\!] &= b \\
\mathcal{C}[\![\ e[f\ t_1 \ldots t_n]\ ]\!] &= f^{\square}\ u_1 \ldots u_l \\
\mathcal{C}[\![\ e[g\ (c\ t_{n+1} \ldots t_{n+k})\ t_1 \ldots t_n]\ ]\!] &= f^{\square}\ u_1 \ldots u_l \\
\mathcal{C}[\![\ e[g\ v\ t_1 \ldots t_n]\ ]\!] &= g^{\square}\ v\ u_1 \ldots u_l
\end{aligned}
$$

where in each case $u_1 \ldots u_k$ (and in the last also v) are the free variables of the term occurring as argument to $\mathcal{C}$.

Example 5. The T-graph for the *last* example turns into the residual term and residual program

$$
\begin{array}{ll}
f_1\ xs & g_1\ []\qquad\qquad \leftarrow B \\
f_1\ xs \leftarrow f_2\ xs & g_1\ (z':zs')\quad\ \leftarrow g_2\ zs'\ z' \\
f_2\ xs \leftarrow f_3\ xs & g_2\ []\ z'\qquad\ \leftarrow z' \\
f_3\ xs \leftarrow g_1\ xs & g_2\ (z'':zs'')\ z' \leftarrow g_2\ zs''\ z''
\end{array}
$$

This program has several intermediate functions which should be unfolded in a *postunfolding phase* yielding a term and program such as

$$
\begin{array}{ll}
g_1\ xs & \\
g_1\ []\qquad\quad \leftarrow B & g_2\ []\ x\qquad\quad \leftarrow x \\
g_1\ (z:zs) \leftarrow g_2\ zs\ z & g_2\ (z:zs)\ x \leftarrow g_2\ zs\ z
\end{array}
$$

In partial evaluation of imperative languages, this operation is called *transition compression.*The new term $g_1\ xs$ is faster than both the original term *last* $(A:B:xs)$ and the above residual term $f_1\ xs$. Removing redundant intermediate functions is a main source for speedups in program specialization.

Generation of residual programs in this manner is possible whenever the generated T-graph is finite. T-graphs constructed as above are not always finite; for this one needs to introduce more sophisticated methods such as *generalizations.*

The correctness of the transformation rules underlying driving was proved in [Tur80a]. A proof of the following result can be recovered from [Sor94c].

Proposition 11 (Correctness of residual programs). *Given a term and program t, p. Assume that the T-graph for t is finite and let t', p' be the residual term and program. Then for every substitution θ mapping all the free variables of t (and thereby t') to values, it holds that $\mathcal{R}[\![\ t\theta\ ]\!] = \mathcal{R}[\![\ t'\theta\ ]\!]$.*[3]

---

[3] The two results should be understood in the context of p and p', respectively. Further, the equality means that either both sides are undefined, or both are defined and denote the same result.

It is worth noting that as stated in the present paper, the transformer propagates only *positive* information, *i.e.* information that a certain variable is bound to a certain value or is instantiated to a certain pattern. We never propagate *negative* information that a variable is *not* bound to a certain value. Negative information only enters the scene once the language contains some kind of an "otherwise" construct. For instance, suppose that our language allowed a "catch-all" clause $g\,x\,v_1\ldots v_n \leftarrow t_{m+1}$ as the last clause of a g-function definition. Then, when transforming a term $g\,v\,b_1\ldots b_n$ leading to $m+1$ branches, one could propagate to the last branch the information that the variable v was not instantiated to any of the patterns of g. For treatments of positive and negative information and the related notion of *perfect trees*, see [Tur80a, Glu93, Sor94c].

4 Logic Programming and Partial Deduction

In this section we consider first-order logic formulas. We shall be concerned only with definite logic programs, *i.e.* programs without negation. The results may therefore diverge in some aspects from Prolog or other logic language dialects which include non-logical extensions such as cut or side-effects.

For simplicity we assume that functors and variables in the logic language follow the conventions of $\mathcal{M}$ constructors and variables, respectively. For the special list functor *Cons* we use the usual short hand notation like $[x_1, x_2 | xs]$ which is slightly different from the short hand notation in $\mathcal{M}$.

The following definition is standard, see [Llo87]. Note that throughout this paper all definitions are stated for the first-literal computation rule.

Definition 12 (SLD-tree). Given program P and goal G, an *SLD-tree* for P, G is a tree of goals defined as follows. (i) the root contains G. (ii) Let $A_1, A_2, \ldots, A_n$, $n > 0$, be the goal in a node N, where the A_i's are atoms. Then, for every standardized apart program clause $A \leftarrow B_1, \ldots, B_m$ such that A and A_1 are unifiable with MGU θ, N has a child $(B_1, \ldots, B_m, A_2, \ldots, A_n)\theta$. ($iii$) nodes containing the empty clause have no children.

Empty nodes are called *success nodes*, and other leaf nodes are called *failure nodes*. Branches ending in success and failure nodes are called success and failure branches, respectively. For a success branch, the composition $\theta_1 \circ \ldots \circ \theta_n$ restricted to the variables of G, where θ_i is computed in the i'th application of (ii) above, is called a *computed answer*.

Example 6. Consider the predicate *last*

$$last([x|xs], r) \leftarrow l(xs, x, r).$$
$$l([], x, x).$$
$$l([z|zs], x, r) \leftarrow l(zs, z, r).$$

The SLD-tree for the goal $last([A, B], r)$ has one branch with the goals

$$last([A, B], r), \quad l([B], A, r), \quad l([], B, r), \quad \square$$

with the computed answer $\{r := B\}$. The SLD-tree for $last([A, B|xs], r)$ is:

$$last([A, B|xs], r)$$
$$|$$
$$l([B|xs], r)$$
$$|$$
$$l(xs, B, r)$$
$$/ \ \ \backslash$$
$$\square \quad l(zs', z', r)$$
$$/ \ \ \backslash$$
$$\square \quad l(zs'', z'', r)$$
$$\vdots$$

where the computed answers are $\{xs := [], r := B\}$, $\{xs := [z'], r := z'\}$, $\{xs := [z', z''], r := z''\}$ *etc.*

An SLD-derivation is either *infinite, successful,* or *failed.* In partial deduction an SLD-tree is allowed to be *incomplete*, in the sense that at any node we are allowed to simply not select any atom and stop the derivation. The following notions are sometimes stated slightly differently, see [Llo91, Kom92].

Definition 13 (Partial SLD-tree, Partial Deduction). A Partial *SLD-tree* for a program P and goal G is a tree of goals defined as follows. (i) the root contains G. (ii) let $A_1, A_2, \ldots, A_n$, $n > 0$, be the goal in a node N, where the A_i's are atoms. If N has any children, then for every standardized apart program clause $A \leftarrow B_1, \ldots, B_m$ such that A and A_1 are unifiable with MGU θ, N has a child $(B_1, \ldots, B_m, A_2, \ldots, A_n)\theta$, and N has no other children. (iii) nodes containing the empty clause have no children.

A branch ending in a node with a leftmost goal that does not unify with the head of any program clause is called a failure node. For non-failure nodes the composition $\theta_1 \circ \ldots \circ \theta_n$ restricted to the variables of G, where θ_i is computed in the i'th application of (ii) above, is called a *computed answer.*

Let G be an atomic goal. The set of all clauses $G\theta \leftarrow G'$ where G' is the end node of a non-failure branch and θ is the computed answer for that branch, is a *partial deduction* of G in P.

For a set $\Gamma = \{G_1 \ldots G_n\}$ of atomic goals, a partial deduction of Γ in P is the union of partial deductions of $G_1 \ldots G_n$ in P. A partial deduction of P with respect to Γ is the result obtained from P by replacing the set of clauses in P whose heads contain a predicate symbol in Γ by a partial deduction of Γ in P.

Definition 14 (Closedness). Let S be a set of first-order formulae and Γ a finite set of atoms. S is Γ-*closed* if each atom in S containing a predicate symbol occurring in Γ is an instance of an atom in Γ.

Example 7. Consider the goal $last([A, B|xs], r)$. A partial SLD-tree is:

$$last([A, B|xs], r)$$
$$|$$
$$l([B|xs], r)$$
$$|$$
$$l(xs, B, r)$$
$$/ \ \backslash$$
$$\square \quad l(zs', z', r)$$

A partial deduction of the *last* program with respect to $\{last([A, B, xs], r)\}$ is:

$$last([A, B], B).$$
$$last([A, B, z'|zs'], r) \leftarrow l(zs', z', r).$$
$$l([], x, x).$$
$$l([z|zs], x, r) \qquad \leftarrow l(zs, z, r).$$

For soundness and completeness of residual programs one has: [Llo91]

Proposition 15. *Let P, G be a program and goal, Γ a finite set of atoms, P' a partial deduction of P with respect to Γ. Then G has computed answer θ in P if G has computed answer in P'. If $P' \cup \{G\}$ is Γ-closed then G has computed answer θ in P' if G has computed answer in P.*

5 Correspondence

We now give a translation from $\mathcal{M}$ programs to logic programs which are equivalent in a certain sense, and we compare supercompilation of the $\mathcal{M}$ programs with partial deduction on their translated logic counterparts. We aim at showing that T-trees and SLD-trees are essentially the same, that T-graphs and partial SLD-trees are essentially the same, and that the mechanisms generating residual programs from finite T-graphs and finite SLD-trees are very similar.

T-trees, T-graphs, SLD-trees, partial SLD-trees

Definition 16 (Tree isomorphisms). A *tree isomorhism* ϕ from a tree T to a tree T' is a mapping from nodes of T to nodes of T' such that: (i) if the root of T is N then the root of T' is $\phi(N)$; (ii) if the different children of N in T are from left to right $N_1 \ldots N_k$ then the different children of $\phi(N)$ in T' are from left to right $\phi(N_1) \ldots \phi(N_k)$.

Definition 17 (Translation from $\mathcal{M}$ to the logic language). Given $\mathcal{M}$ term and program t, p. Let r be a specific chosen variable, let y in each clause be a fresh variable, and define a translation $\underline{\bullet}$ from terms to lists of atoms as follows:

$\underline{b}$	$\equiv r = b.$
$\underline{e[f \ b_1 \ldots b_n]}$	$\equiv f(b_1, \ldots, b_n, y), \underline{e[y]}$
$\underline{e[g \ (c \ b_{n+1} \ldots b_{n+k}) \ b_1 \ldots b_n]}$	$\equiv g((c \ b_{n+1}, \ldots, b_{n+k}), b_1, \ldots, b_n, y), \underline{e[y]}$
$\underline{e[g \ v \ b_1 \ldots b_n]}$	$\equiv g(v, b_1, \ldots, b_n, y), \underline{e[y]}$

A translation on programs $\underline{p}$ is given by the following translation on each definition:

$$\underline{f \ v_1 \ldots v_n \leftarrow t} \equiv P_f(v_1, \ldots, v_n, r) : -\underline{t}$$

and

$$
\begin{array}{ccc}
g \ p_1 \ v_1 \ldots v_n \ \leftarrow t_1 & & P_g(p_1, v_1, \ldots, v_n, r) \ : - \ \underline{t_1} \\
\vdots & \equiv & \vdots \\
\underline{g \ p_m \ v_1 \ldots v_n \leftarrow t_m} & & P_g(p_m, v_1, \ldots, v_n, r) : - \ \underline{t_m}
\end{array}
$$

After applying $\bullet$, turn every clause of form $h(a_1, \ldots, a_n, r) \leftarrow r = b.$ into $h(a_1, \ldots, a_n, b).$ and those of form $h(a_1, \ldots, a_n, r) \leftarrow \ldots, h'(a'_1, \ldots, a'_{n'}, y), r = y.$ into $h(a_1, \ldots, a_n, r) \leftarrow \ldots, h'(a'_1, \ldots, a'_{n'}, r).$

Example 8. As an example the reader may verify that the translation of the *last* function from Example 1 gives the *last* predicate in Example 6.

Proposition 18. *For any ground $\mathcal{M}$ term t and program p, the SLD-tree for $\underline{t}, \underline{p}$ has exactly one branch.*

Example 9. This was the case with the goal $last([A, B], r)$ in Example 6.

The core of this phenomenon is that atomic goals are always called with all variables completely instantiated, except for the variable in the last argument, and this variable is instantiated after the goal is satisfied. The instantiation of this variable represents the returning of a value by a function in $\mathcal{M}$ programs.

We can now state that our translation preserves semantics.

Proposition 19. *Given a ground $\mathcal{M}$ term t and a program p. Let T be the I-sequence for t, T' be the SLD tree for $\underline{t}, \underline{p}$. It then holds that either (i) both T and T' are infinite, or (ii) both are finite and the last node of T contains the term a and T' has the computed answer $\{r := a\}$ in its single branch.*

Example 10. As an example, this holds for the transformation tree for *last* $[A, B]$ and the goal $last([A, B], r)$, see Examples 1 and 6.

We finally have the desired equivalence between T-trees and SLD-trees.

Proposition 20. *Given $\mathcal{M}$ term and program t, p. Let T be the T-tree for t, p, and T' the SLD-tree for $\underline{t}, \underline{p}$. Then there is a tree isomorphism ϕ from T to T' that maps a node N containing the term t to a node N' containing the empty clause if t is a constructor term, and $\underline{t}$ otherwise.*

Example 11. As an example, this holds for the transformation tree for *last* $(A : B : xs)$ and the goal $last([A, B|xs], r)$, see Examples 2 and 6.

Just like T-trees and SLD-trees are the same objects in different disguises, T-graphs and partial SLD-trees are strongly related.

Example 12. Compare the T-graph in Example 4 with the partial SLD-tree in Example 7. The only essential difference between the two is that the T-graph has an extra node (containing z') and two extra arcs (one to z' and one from the lower right most node to itself).

The difference stems from the different ways of folding and generating residual program, see the next subsection. Here it suffices to note that there is an explicit folding scheme in T-graphs, *i.e.* a given way of deciding when to stop developing a branch, whereas this is left open in the case of partial SLD-trees.

We now proceed from T-graphs and partial SLD-trees to residual functional and logic programs, respectively.

Residual programs and partial deductions

Example 13. Compare the first of the functional programs in Example 5 with the residual logic program in Example 7. Note that the functional program has some intermediate functions, whereas the logic program has none; transition compression has already been done in the partial deduction. The transition compression comes from the fact that every residual clause has right hand side the end node of the branch, whereas in the functional program, residual functions are made for every node in the branch. In the partial deduction the MGU's along the branch are substituted into the original goal (which becomes a left hand side) whereas in the functional program one has a simple pattern for every function definition.

There are some subtleties involved with collecting subsequent instantiations in a nested pattern in the functional case. For instance, compare the following two programs, both of which are slightly out of $\mathcal{M}$ syntax.

$$
\begin{array}{ll}
f\ (S\ x) \leftarrow g\ x & \qquad f\ (S\ Z) \leftarrow Z \\
f\ y \quad\ \leftarrow y & \qquad f\ y \quad\ \leftarrow y \\
g\ Z \quad\ \leftarrow Z &
\end{array}
$$

The transformation of the former into the latter seems reasonable, but note that the term $f\ x$ is undefined for $x := S\ (S\ Z)$ for the first program, but not for the second program; collecting instantiations in a nested pattern can extend the domain of a function.

In partial deduction one can turn a version of the former into a version of the latter program:

$$
\begin{array}{ll}
f(S\ x, r) \leftarrow g(x, r). & \qquad f(S\ Z, Z). \\
f(y, y). & \qquad f(y, y). \\
g(Z, Z). &
\end{array}
$$

The two programs have the same computed answers, but some branches are pruned earlier in the SLD-tree for the latter program. Related problems are explained in Section 9.5 of [Sor94b] following [Tur80a].

Example 14. Now let us instead compare the second of the functional programs from Example 5 with the same logic program.

First note that the functional residual program has retained none of the functions from the original functional program from Example 1. This always holds. In T-graphs one only stops developing a branch if a term is reached which has previously been seen in the same branch. Because of this, every right hand side in the residual program stemming from some node N in the T-graph becomes either a constructor term, a call to the function stemming from the next term in the branch, or a call to a function stemming from a previous node in the same branch. But note also that the new funtion g_2 is just a copy of the original function l.

As the example shows, a partial deduction may retain some of the clauses from the original program. So one needs to ensure that one retains all clauses from the original program that are (possibly) called in right hand sides of the new residual clauses or in the right hand sides of other clauses from the original program that are retained. This is what the closedness condition ensures.

One can devise an algorithm for partial deduction satisfying the closedness condition, see the *Basic Algorithm* in [Gal93]. This algorithm constructs a partial SLD-tree with certain points for looping back (the end nodes of every intermediate partial tree), similar to the construction of T-graphs.

Continuing the comparison in the latter example above we see also another difference between the functional and logic residual program: most of the constants from the original program are gone in the residual functional program, but remain in the residual logic program. In partial deduction one needs an extra *renaming* phase with *structure specialization* to do this, see [Kom92, Gal93]. A similar approach for specialization is taken in [Jon94].

Proietti and Pettorossi [Pro93] argue that renaming should be incorporated into a partial deducer directly, and that partial deduction be carried out in a unfold/fold transformation (of logic programs) style which is similar to our driving.

6 Applications

In this section we consider two applications of the correspondence.

Logic programming by driving

One can perceive a partial deduction as a finite representation of a possibly infinite set of computed answers for the original goal and program. The similarity between positive supercompilation and partial deduction then suggests that one should be able to obtain logic programming effects by positive supercompilation of functional programs. We now show how this is done.

Consider the predicate *connect*:

$$connect(x, y, []) \leftarrow flight(x, y).$$
$$connect(x, y, [z|zs]) \leftarrow flight(x, z), connect(z, y, zs).$$

$$flight(Vienna, Paris).$$
$$flight(Vienna, Rome).$$
$$flight(Rome, Paris).$$
$$flight(Paris, Copenhagen).$$

The goal $\leftarrow connect(z, y, zs)$ is satisfiable if there are flights (x_1, x_2), (x_2, x_3) ... (x_{n-2}, x_{n-1}), (x_{n-1}, x_n) such that x is x_1, y is x_n and zs is $[x_2, \ldots x_{n-1}]$. For instance, $\leftarrow connect(Vienna, Copenhagen, via)$ has computed answers: $via := [Paris], via = [Rome, Paris]$.

On the other hand, the goal $\leftarrow connect(Vienna, Copenhagen, via)$ has partial deduction:

$$connect(Vienna, Copenhagen, [Paris]).$$
$$connect(Vienna, Copenhagen, [Rome, Paris]).$$

The goal $\leftarrow connect(Vienna, Copenhagen, via)$ has the same computed answers in both programs, but in the latter program the fact is so obvious that one could say that the residual program is a very concrete representation of the set of computed answers of the original program—the residual program is basically a case dispatch with an entry for every computed answer of the original program.

Since there are no free variables in the body of any predicates in the original logic program, it is straight-forward to turn it into a $\mathcal{M}$ program:

$connect\ x\ y\ Nil$	$\leftarrow flight\ x\ y$	$flight\ Vienna\ Paris$	$\leftarrow True$
$connect\ x\ y\ (z : zs)$	$\leftarrow g\ (flight\ x\ z)\ z\ y\ zs$	$flight\ Vienna\ Rome$	$\leftarrow True$
		$flight\ Rome\ Paris$	$\leftarrow True$
$g\ True\ z\ y\ zs$	$\leftarrow connect\ z\ y\ zs$	$flight\ Paris\ Copenhagen$	$\leftarrow True$
$g\ False\ z\ y\ zs$	$\leftarrow False$	$flight\ x\ y$	$\leftarrow False$

If, for the term *connect Vienna Copenhagen via*, we construct the transformation graph and then construct the residual program from this graph and apply postunfolding, we get the following term and program.[4]

$$c^1\ via$$

$c^1\ Nil$	$\leftarrow False$	$f^1\ Paris\ zs$	$\leftarrow c^2\ zs$
$c^1\ (z : zs)$	$\leftarrow f^1\ z\ zs$	$f^1\ Rome\ zs$	$\leftarrow c^3\ zs$
		$f^1\ x\ zs$	$\leftarrow False$
$c^2\ Nil$	$\leftarrow True$	$f^2\ Copenhagen\ zs$	$\leftarrow c^c\ zs$
$c^2\ (z : zs)$	$\leftarrow f^2\ z\ zs$	$f^2\ x\ zs$	$\leftarrow False$
$c^3\ Nil$	$\leftarrow False$		
$c^3\ (z : zs)$	$\leftarrow False$		
$c^4\ Nil$	$\leftarrow False$	$f^4\ Paris\ zs$	$\leftarrow c^2\ zs$
$c^4\ (z : zs)$	$\leftarrow f^4\ z\ zs$	$f^4\ z\ zs$	$\leftarrow False$

---

[4] For the sake of brevity we have cheated slightly in that the call to c^2 in f^4 should actually be a call to a copy d^2 of c^2 where the subfunctions of d^2 are copies of the subfunctions of c^2.

If, in addition, the positive compiler would collect subsequent instantiations in one nested pattern, and if it would also perform the optimization of checking whether different branches yielded the same result, and in this case omit the corresponding instantiations or tests, we could get:

$$c\,[Paris] \qquad \leftarrow True$$
$$c\,[Rome, Paris] \leftarrow True$$
$$c\,x \qquad\qquad \leftarrow False$$

There is an implementation of the supercompiler which returns this result.

The relation of program specialization, theorem proving, and program inversion has been discussed more thoroughly in the context of metacomputation [Tur80b, Tur82, Abr91, Rom91, Glu94]. As one of the first examples of driving, it was shown that it is capable of inverting an algorithm for binary addition and perform binary subtraction [Tur72, Glu90].

Non-termination patterns

We will close the comparison by showing the correspondence of non-termination patterns encountered in program specialization by driving and partial deduction. It should be obvious that the techniques for ensuring termination in partial deduction are closely related to the methods in the functional world, and that both sides may benefit from each other. However, an in-depth comparison of termination strategies is beyond the scope of this paper.

Example 15 (The accumulating parameter). Consider the following $\mathcal{M}$ program and term which tests whether the third argument is the list reversal of the first.

$$isrev\ v\ []\ w$$
$$isrev\ (x:xs)\ ys\ zs \leftarrow isrev\ xs\ (x:ys)\ zs$$
$$isrev\ []\ yz\ zs \qquad \leftarrow equal\ yz\ zs$$

where we leave out the details of *equal*. Although the function is written in $\mathcal{M}$ it has a direct correspondence to the standard, efficient *reverse* predicate in logic programming.

Positive supercompilation of *isrev* $v\ []\ w$ does not terminate, because the following progressively larger terms are encountered: *isrev* $v\ []\ w$, *isrev* $v'\ [x']\ w$, *isrev* $v''\ [x'', x']\ w$, *etc.* The folding scheme is not able to construct a finite T-graph because of this.

Accumulating parameters are a typical method for building intermediate structures in logic programs, resulting in the same non-termination pattern in the case of partial deduction. In fact similar problems occur in partial deduction, see [Gal93].

The problem of *the accumulating parameter* is *one* non-termination pattern. This and the problem of *the obstructing function call* were characterized by Chin for deforestation [Chi93]. A third one, the problem of *the accumulating side effect* also occurs in driving and partial deduction, see [Sor93, Sor94b]. The framework for preventing termination problems, *generalizations*, is also very similar in both worlds, see [Gal93, Sor93].

7 Conclusion and Future Work

Driving is, in a sense, closer to partial deduction than to partial evaluation in functional languages. This relation of seemingly different activities such as program specialization, theorem proving, or program inversion shows how closely these activities are interconnected, and that one should not consider them separately. On the contrary, we have not doubt that different transformation methods in different paradigms can benefit from each other. As stated earlier a next step will be, *e.g.* to investigate methods for ensuring termination.

References

[Abr91] S. M. Abramov. Metacomputation and Logic Programming. In *Programmirovanie.* 1991 (3): 31-44 (in Russian).

[Chi93] W.-N. Chin. Safe Fusion of Functional Expressions II: Further Improvements. In *Journal of Functional Programming.* 1994 (to appear).

[Gal93] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation. (Copenhagen, Denmark).* 88-98, ACM Press 1993.

[Glu90] R. Glück & V. F. Turchin. Application of Metasystem Transition to Function Inversion and Transformation. In *Proceedings of the ISSAC '90 (Tokyo, Japan).* 286-287, ACM Press 1990.

[Glu93] R. Glück & And. Klimov. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. In *P. Cousot et al. (Eds.), Static Analysis, Proceedings.* Lecture Notes in Computer Science, Vol. 724, 112-123, Springer-Verlag 1993.

[Glu94] R. Glück, And. Klimov. Metacomputation as a Tool for Formal Linguistic Modeling. In *R. Trappl (ed.), Cybernetics and Systems '94.* Vol. 2, 1563-1570, World Scientific: Singapore 1994.

[Hol91] C. K. Holst. Partial Evaluation is Fuller Laziness. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation. (New Haven, Connecticut).* Sigplan Notices vol. 26, No. 9, September 1991, 223-233, ACM Press 1991.

[Jon88] N. D. Jones. Automatic Program Specialization: a Re-Examination from Basic Principles. In *D. Bjørner, A. P. Ershov, & N. D. Jones (eds.), Partial Evaluation and Mixed Computation.* 225-282, North-Holland 1988.

[Jon93] N. D. Jones, C. Gomard & P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International. 1993

[Jon94] N. D. Jones. The Essence of Program Transformation by Partial Evaluation and Driving. In *N. D. Jones and M. Hagiya and M. Sato (eds.), Logic, Language and Computation.* Lecture Notes in Computer Science, Vol. 792, 206-224, Springer-Verlag 1994.

[Kom81] J. Komorowski. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation.* Linkøping University. Dissertation, 1981.

[Kom92] J. Komorowski. An Introduction to Partial Deduction. In *A. Pettorossi (ed.), Meta-Programming in Logic. Proceedings. (Uppsala, Sweden).* Lecture Notes in Computer Science, Vol. 649, 49-69, Springer-Verlag 1992.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming. Second, extended edition.* Springer-Verlag, 1987.

[Llo91] J. W. Lloyd & J. C. Shepherdson. Partial Evaluation in Logic Programming. In *Journal of Logic Programming.* 11(3-4): 217-242, 1991.

[Pro93] M. Proietti & A. Pettorossi. The Loop Absorption and the Generalization Strategies for the Development of Logic Programs and Partial Deduction. In *Journal of Logic Programming.* 16(1&2): 123-161, 1993.

[Rom91] A. Y. Romanenko. Inversion and Metacomputation. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation. (New Haven, Connecticut).* 12-22, ACM Press 1991.

[Sor93] M. H. Sørensen. A New Means of Ensuring Termination of Deforestation with an Application to Logic Programming. In *Workshop of the Global Compilation Workshop in conjunction with the International Logic Programming Symposium.* Vancouver, Canada, October, 1993.

[Sor94a] M. H. Sørensen. A Grammar-Based Data-Flow Analysis to stop Deforestation. In *S. Tison (ed.), Trees in Algebra and Programming - CAAP '94. Proceedings. (Edinburgh, Scotland).* Lecture Notes in Computer Science, Vol. 787, 335-351, Springer-Verlag 1994.

[Sor94b] M. H. Sørensen. *Turchin's Supercompiler Revisited. An Operational Theory of Positive Information Propagation.* Master's Thesis, Department of Computer Science, University of Copenhagen, 1994, Technical report 94/7.

[Sor94c] M. H. Sørensen, R. Glück & N. D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. In *D. Sannella (ed.), Programming Languages and Systems - ESOP '94. Proceedings. (Edinburgh, Scotland).* Lecture Notes in Computer Science, Vol. 788, 485-500, Springer-Verlag 1994.

[Tur72] V. 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).* 31-42, 1972 (in Russian).

[Tur80a] V. F. Turchin. *The Language Refal, the Theory of Compilation and Metasystem Analysis.* Courant Computer Science Report No. 20, New York University 1980.

[Tur80b] V. F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimizattion. In *J. W. de Bakker & J. van Leeuwen (Eds.), Automata, Languages and Programming (Noordwijkerhout, Netherlands).* Lecture Notes in Computer Science, Vol. 85, 645-657, Springer-Verlag 1980.

[Tur82] V. F. Turchin, R. Nirenberg & D. Turchin. Experiments with a Supercompiler. In *Conference Record of the ACM Symposium on Lisp and Functional Programming.* 47-55, ACM Press 1982.

[Tur86] V. F. Turchin. The Concept of a Supercompiler. In *ACM TOPLAS.* 8(3), 292-325, 1986.

[Wad90] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *Theoretical Computer Science.* 73: 231-248, 1990.

This article was processed using the LaTeX macro package with LLNCS style