Implementing circularity using partial evaluation

Julia L. Lawall[⋆]

DIKU, University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen, Denmark

Abstract. Complex data dependencies can often be expressed concisely by defining a variable in terms of part of its own value. Such a circular reference can be naturally expressed in a lazy functional language or in an attribute grammar. In this paper, we consider circular references in the context of an imperative C-like language, by extending the language with a new construct, *persistent variables*. We show that an extension of partial evaluation can eliminate persistent variables, producing a staged C program. This approach has been implemented in the Tempo specializer for C programs, and has proven useful in the implementation of run-time specialization.

1 Introduction

In compilation and program transformation, the treatment of a subcomponent of a block of code often depends on some global properties of the code itself. A compiler needs to know whether the source program ever uses the address of a local variable, to decide whether the variable must be allocated on the stack [1]. A partial evaluator needs to know whether a dynamic (but non-side-effecting) expression is referred to multiple times, to decide whether the expression should be named using a let expression [4, 23]. In the context of run-time specialization, we have found that optimizing the specialized code based on its total size and register usage can significantly improve its performance [15]. Such programs can often be efficiently implemented as multiple phases, where early phases collect information and later phases perform the transformation. This organization, however, distributes the treatment of each subcomponent of the input across the phases, which can introduce redundancy, and complicate program understanding and maintenance.

In a lazy language, we can implement these examples in a single traversal using recursive declarations, as investigated by Bird [3]. The canonical example of such a *circular* program is “repmin,” which reconstructs a tree, such that the value at each leaf is the least value at any leaf in the original tree. While repmin can be implemented by first traversing the tree to detect the least value, and

[⋆] E-mail: `julia@diku.dk`. This research was partially supported by the Danish Natural Science Research Council (*PLT* project).

then traversing the tree again to construct the result, the program can also be expressed in a lazy language as follows [3]:

```
data Tree = Tip Int | Fork Tree Tree

rm t = fst p
  where p = repmin t (snd p)

repmin (Tip n) m = (Tip m, n)

repmin (Fork L R) m = (Fork t1 t2, min m1 m2)
  where (t1, m1) = repmin L m
    and (t2, m2) = repmin R m
```

The variable `p` in `rm` represents a value that is the result of the traversal of the tree, but that is used in computing that result as well. Here, a lazy evaluation strategy suffices to order the computations such that the components of `p` are determined before they are used.

Nevertheless, the use of a lazy language is not always appropriate. To resolve this dilemma, we propose a language extension, *persistent variables*, that can describe circular references in an imperative language. Using this facility, we can implement repmin imperatively as follows, where the persistent variable `minval` implements the circular reference to `p` in the functional implementation:[1]

```
typedef struct ans {
  int mn;
  Tree *tree;
} Ans;

Tree *rm(Tree *t) {
  persistent int minval;
  Ans a;

  repmin(t, pread(minval), &a);
  pwrite(minval,a.mn);
  return a.tree;
}
```

```
void repmin(Tree *t, int m, Ans *a) {
  Ans a1, a2;
  if (t->type == Fork) {
    repmin(t->left,  m, &a1);
    repmin(t->right, m, &a2);
    a->mn   = min(a1.mn,a2.mn);
    a->tree = mkFork(a1.tree,a2.tree);
  }
  else /* (t->type == Tip) */ {
    a->mn   = t->tipval;
    a->tree = mkTip(m);
  }
}
```

This implementation uses `pread` ("persistent read") to reference the final value to which `minval` is initialized using `pwrite` ("persistent write"). To execute the program, we must first transform it such that it initializes `minval` before any reference. In this paper, we show how to use partial-evaluation technology to perform this transformation.

Traditionally, partial evaluation specializes a program with respect to a subset of its inputs. The program is evaluated in two stages: The first stage performs the *static* computations, which depend only on the known input. When

[1] The structure `Ans` is used to return multiple values from the function `repmin`.

the rest of the input is available, the second stage evaluates the remaining *dynamic* computations, producing the same result as the original program. With minor extensions, we can use this framework to eliminate circularity by simply considering computations that depend on the persistent variables to be dynamic, and the other computations to be static. For example, in the above implementation of repmin, the construction of each leaf of the output tree depends on the value of the persistent variable `minval`, *via* the parameter `m` of `repmin`, and is thus dynamic. The calculation of the least value in the tree depends only on the input tree, and is thus static. The staging performed by partial evaluation permits the persistent variables that only depend on static information to be initialized in the static phase and read in the dynamic phase. We have implemented this approach in the Tempo partial evaluator for C programs, developed in the Compose group at IRISA [6].

This implementation of circularity leads naturally to *incremental specialization* [7, 17]; if the value of a persistent variable depends on that of another persistent variable, partial evaluation must be iterated. If there are recursive dependencies among the persistent variables, however, the program cannot be treated by our approach (*cf.* Section 6.3).

The rest of this paper is organized as follows. Section 2 describes partial evaluation in more detail. Section 3 presents the implementation of persistent variables in the context of a partial evaluator for a simple imperative language. Section 4 gives a semantics of the language with persistent variables and shows that partial evaluation of a program preserves its semantics. Section 5 compares our partial evaluation-based approach to related techniques in the implementation of attribute grammars. Section 6 provides some examples of the use of persistent variables. Section 7 describes related work, and Section 8 concludes.

2 Specialization using partial evaluation

Partial evaluation uses interprocedural constant propagation to specialize a program with respect to some of its inputs. We use *offline* partial evaluation, in which each expression is annotated as static or dynamic by a preliminary *binding-time analysis* phase. Two kinds of specialization can be performed in this framework: *program specialization* and *data specialization.* Program specialization transforms a program into an optimized implementation based on the results of evaluating the static subexpressions [11]. Static subexpressions are replaced by their values, static conditionals are reduced, and static loops are unrolled. Data specialization separates a program into two stages, known as the *loader* and the *reader*, before the static data is available [2]. The loader stores the values of the static subexpressions in a data structure known as a *cache.* The reader has the form of the source program, but with the static subexpressions replaced by cache references. Because the loader and reader are independent of the static data, conditionals are not reduced and loops are not unrolled.

We implement persistent variables in the context of data specialization. Persistent variables and data specialization fit together well, because the data specialization cache is a natural means to transmit the values of persistent variables from the static phase to the dynamic phase.

3 Data specialization and persistent variables

We now define data specialization for a simple imperative language with persistent variables. Treating a richer language is straightforward; the implementation allows the full subset of C accepted by Tempo [6], including pointers and recursive functions.

3.1 Source language

A program consists of declarations d and a statement s, defined as follows:

$$\begin{array}{lcl}
d \in \textit{Declaration} & ::= & \texttt{int x} \mid \texttt{persistent int p} \\
s \in \textit{Statement} & ::= & \texttt{x}\,\texttt{=}\,e \mid \texttt{pwrite(p,}\,e\texttt{)} \mid \texttt{if (}e\texttt{)}\; s_1 \;\texttt{else}\; s_2 \\
 & & \mid \texttt{while (}e\texttt{)}\; s \mid \texttt{\{}s_1\,\texttt{;}\,\ldots\,\texttt{;}\,s_n\texttt{\}} \\
e \in \textit{Expression} & ::= & c \mid \texttt{x} \mid e_1 \;op\; e_2 \mid \texttt{pread(p)} \\
\texttt{x} \in \textit{Variable} & & \\
\texttt{p} \in \textit{Persistent variable} & & \textit{Variable} \cap \textit{Persistent variable} = \emptyset
\end{array}$$

A persistent variable can only appear as the first argument of `pread` or `pwrite`. Thus, a persistent variable is essentially a label, rather than a first-class value.

3.2 Binding-time annotation

Binding times are static, S, and dynamic, D, where $\mathsf{S} \sqsubset \mathsf{D}$. The language of binding-time annotated declarations $\hat{d}$, statements $\hat{s}$, and expressions $\hat{e}$ is defined as follows:

$$\begin{array}{lcl}
b \in \textit{Binding time} & = & \{\mathsf{S}, \mathsf{D}\} \\
\hat{d} \in \textit{BT-Declaration} & ::= & \texttt{int x}^b \mid \texttt{persistent int p}^b \\
\hat{s} \in \textit{BT-Statement} & ::= & \texttt{x}\,\texttt{=}^b\,\hat{e}^{b'} \mid \texttt{pwrite(p}^b\texttt{,}\,\hat{e}^{b'}\texttt{)} \mid \texttt{if (}\hat{e}^b\texttt{)}\;\hat{s}_1\;\texttt{else}\;\hat{s}_2 \\
 & & \mid \texttt{while (}\hat{e}^b\texttt{)}\;\hat{s} \mid \texttt{\{}s_1\,\texttt{;}\,\ldots\,\texttt{;}\,s_n\texttt{\}} \\
\hat{e} \in \textit{BT-Expression} & ::= & c \mid \texttt{x} \mid \hat{e}_1^{b_1}\;op\;\hat{e}_2^{b_2} \mid \texttt{pread(p}^b\texttt{)}
\end{array}$$

Figure 1 presents inference rules that specify binding-time annotations for a program, based on an environment Γ mapping variables and persistent variables to binding times. In the annotation of a program, $\Gamma(d)$ represents the binding time associated by Γ to the variable declared by d. The annotation of a statement s is described by $\Gamma, b_c \vdash_{\mathrm{s}} s : \hat{s}$, where the binding time b_c is the least upper bound of the binding-times of the enclosing conditional and loop tests. The annotation of an expression e is described by $\Gamma \vdash_{\mathrm{e}} e : \hat{e}^b$. Annotations can be automatically inferred using standard techniques [11].

The rules of Figure 1 treat statements and expressions as follows. The annotation of an assignment statement is determined by the binding time of the

Programs:

$$\frac{\Gamma, \mathsf{S} \vdash_{\mathrm{s}} s : \hat{s}}{\Gamma \vdash_{\mathrm{p}} d_1 \ldots d_n s : \hat{d}_1^{\Gamma(d_1)} \ldots \hat{d}_n^{\Gamma(d_1)} \hat{s}}$$

Statements:

$$\frac{\Gamma[\mathsf{x} \mapsto b] \vdash_{\mathrm{e}} e : \hat{e}^{b'} \qquad b \sqsupseteq b_c \sqcup b'}{\Gamma[\mathsf{x} \mapsto b], b_c \vdash_{\mathrm{s}} \mathtt{x}\,\mathtt{=}\,e : \mathtt{x}\,\mathtt{=}^{b}\, \hat{e}^{b'}} \qquad \frac{\Gamma[\mathsf{p} \mapsto b] \vdash_{\mathrm{e}} e : \hat{e}^{b'} \qquad b \sqsupseteq b_c \sqcup b'}{\Gamma[\mathsf{p} \mapsto b], b_c \vdash_{\mathrm{s}} \mathtt{pwrite(p,}\ e\mathtt{)} : \mathtt{pwrite(}\hat{\mathtt{p}}^{b}\mathtt{,}\ \hat{e}^{b'}\mathtt{)}}$$

$$\frac{\Gamma \vdash_{\mathrm{e}} e : \hat{e}^{b} \qquad \Gamma, b_c \sqcup b \vdash_{\mathrm{s}} s_1 : \hat{s}_1 \qquad \Gamma, b_c \sqcup b \vdash_{\mathrm{s}} s_2 : \hat{s}_2}{\Gamma, b_c \vdash_{\mathrm{s}} \mathtt{if}\ \mathtt{(}e\mathtt{)}\, s_1\, \mathtt{else}\, s_2 : \mathtt{if}\ \mathtt{(}\hat{e}^{b}\mathtt{)}\, \hat{s}_1\, \mathtt{else}\, \hat{s}_2}$$

$$\frac{\Gamma \vdash_{\mathrm{e}} e : \hat{e}^{b} \qquad \Gamma, b_c \sqcup b \vdash_{\mathrm{s}} s : \hat{s}}{\Gamma, b_c \vdash_{\mathrm{s}} \mathtt{while}\ \mathtt{(}e\mathtt{)}\, s : \mathtt{while}\ \mathtt{(}\hat{e}^{b}\mathtt{)}\, \hat{s}} \qquad \frac{\Gamma, b_c \vdash_{\mathrm{s}} s_1 : \hat{s}_1 \qquad \ldots \qquad \Gamma, b_c \vdash_{\mathrm{s}} s_n : \hat{s}_n}{\Gamma, b_c \vdash_{\mathrm{s}} \mathtt{\{}s_1\mathtt{;}\ \ldots\ \mathtt{;}s_n\mathtt{\}} : \mathtt{\{}\hat{s}_1\mathtt{;}\ \ldots\ \mathtt{;}\hat{s}_n\mathtt{\}}}$$

Expressions:

$$\Gamma \vdash_{\mathrm{e}} c : c^{\mathsf{S}} \qquad \Gamma[\mathsf{x} \mapsto b] \vdash_{\mathrm{e}} \mathsf{x} : \mathsf{x}^{b} \qquad \frac{\Gamma \vdash_{\mathrm{e}} e_1 : \hat{e}_1^{b_1} \qquad \Gamma \vdash_{\mathrm{e}} e_2 : \hat{e}_2^{b_2}}{\Gamma \vdash_{\mathrm{e}} e_1\, op\, e_2 : (\hat{e}_1^{b_1}\, op\, \hat{e}_2^{b_2})^{b_1 \sqcup b_2}}$$

$$\Gamma[\mathsf{p} \mapsto b] \vdash_{\mathrm{e}} \mathtt{pread(p)} : \mathtt{pread(p}^{b}\mathtt{)}^{\mathsf{D}}$$

Fig. 1. Binding-time analysis

assigned variable, which must be greater than or equal to the binding time of the right-hand side expression and the binding times of the enclosing conditional and loop tests (b_c). The annotation of a `pwrite` statement is similarly constrained. In the annotation of a conditional statement, the least upper bound of b_c and the binding time of the test expression is propagated to the analysis of the branches. The annotation of a loop is similar. The result of a `pread` expression is always dynamic; the binding time of its argument is obtained from the environment. The treatment of the other constructs is straightforward.

This analysis is flow-insensitive and does not allow static assignments under dynamic conditionals and loops. These restrictions can be removed for ordinary variables by existing techniques [9]. Nevertheless, persistent variables must be flow-insensitive, to ensure that every assignment to a persistent variable occurs before any access. An implementation strategy is to perform the binding-time analysis in two phases. The first phase annotates all expressions except persistent variables, using a flow-sensitive analysis. The second phase annotates each persistent variable with the least upper bound of the binding times of all `pwrite` statements at which it is assigned. This second phase does not affect the binding times of other terms, because the binding time of a `pread` expression is dynamic, independent of the binding-time of the associated persistent variable, and a `pwrite` statement has no return value.

3.3 Data specialization

Data specialization stages the source program into a loader and a reader that communicate *via* a cache, which we represent as an array. We thus extend the language with constructs for manipulating cache elements:

$$\begin{array}{l}
s \in \mathit{Statement} ::= \ldots \mid \texttt{\{Cache}\,\mathsf{x}\texttt{;}\,s\texttt{\}} \mid \texttt{*}e_1 \texttt{=} e_2 \\
e \in \mathit{Expression} ::= \ldots \mid \texttt{*}e \\
\mathsf{x} \in \mathit{Variable} \cup \{\texttt{cache}, \texttt{tmp}\}
\end{array}$$

The statement $\texttt{\{Cache}\,\mathsf{x}\texttt{;}\,s\texttt{\}}$ is a block that declares a pointer into the cache. Two such pointers are `cache`, which is initialized to an external array, and `tmp`, which is used in the translation of dynamic conditionals and loops. The indirect assignment $\texttt{*}e_1 \texttt{=} e_2$ and the indirect reference $\texttt{*}e$ initialize and access cache elements, respectively. In practice, bounds checks and casts into and out of a generic cache-element type are added as needed. We refer to this language as the *target language*, and the sublanguage of Section 3.1 as the *source language*.

Figure 2 presents the transformation rules for data specialization of statements and expressions. The transformation of a statement is described by $i \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s} : \langle l, r, i' \rangle$, where i is the offset from `cache` of the next free cache entry, l is a statement representing $\hat{s}$ in the loader, r is a statement representing $\hat{s}$ in the reader, and i' is the offset from `cache` of the next free cache entry after executing either l or r. By keeping track of the cache offset i, we reduce the number of assignments to the cache pointer. The transformation of an expression is described by $i \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}^b : \langle l, v, r, i' \rangle$, where i and i' represent cache offsets as for statements, l is a statement initializing the cache with the values of the static subexpressions of $\hat{e}^b$, v is an expression to use in the loader to refer to the static value of $\hat{e}^b$ if it has one, and r is an expression representing the value of $\hat{e}^b$ for use in the reader. In the definition of the transformation, e is the result of removing all binding-time annotations from the annotated expression $\hat{e}^b$.

The transformation treats statements and expressions as follows. We use cache entries to implement static persistent variables. Thus, $\texttt{pwrite(p}^{\mathsf{S}}\texttt{,}\,\hat{e}^b\texttt{)}$ is translated into an indirect assignment to the entry allocated to `p`. This assignment is placed in the loader. Similarly, $\texttt{pread(p}^{\mathsf{S}}\texttt{)}^{\mathsf{D}}$ is translated into an indirect reference to the corresponding entry. This reference is placed in the reader. References and assignments to a dynamic persistent variable are placed in the reader. The treatment of the remaining constructs is standard [5, 14]. We include the translation of static conditionals here to give a flavor of the adjustments to the cache pointer needed to implement branching control flow constructs. The complete treatment of conditionals and while loops is presented in Appendix A.

We conclude with $\mathsf{ds}[\![p]\!]_{\Gamma}$, the transformation of a program p with respect to a binding-time environment Γ. Let $\hat{d}_1^{b_1} \ldots \hat{d}_n^{b_n} \hat{s}$ be the result of annotating p with respect to Γ. Suppose that the first m declarations of p declare the static persistent variables $\mathsf{p}_1, \ldots, \mathsf{p}_m$. If $m \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s} : \langle l, r, i' \rangle$, then $\mathsf{ds}[\![p]\!]_{\Gamma}$ is:

```
d_{m+1} ... d_n
{
   Cache cache, p_1, ..., p_m;
   cache = cache_start; p_1 = cache;...;p_m = cache + m - 1;
   l; cache = cache_start; r
}
```

Statements:

$$i \vdash^{\mathrm{s}}_{\mathrm{d}} \mathtt{x} =^{\mathsf{S}} \hat{e}^{b} : \langle \mathtt{x} = e, \mathtt{\{\}}, i\rangle \qquad\qquad i \vdash^{\mathrm{s}}_{\mathrm{d}} \mathtt{pwrite(p}^{\mathsf{S}}\mathtt{,}\ \hat{e}^{b}\mathtt{)} : \langle \mathtt{*p}\ = e, \mathtt{\{\}}, i\rangle$$

$$\frac{i \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}^{b} : \langle l, v, r, i'\rangle}{i \vdash^{\mathrm{s}}_{\mathrm{d}} \mathtt{x} =^{\mathsf{D}} \hat{e}^{b} : \langle l, \mathtt{x} = r, i'\rangle} \qquad\qquad \frac{i \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}^{b} : \langle l, v, r, i'\rangle}{i \vdash^{\mathrm{s}}_{\mathrm{d}} \mathtt{pwrite(p}^{\mathsf{D}}\mathtt{,}\ \hat{e}^{b}\mathtt{)} : \langle l, \mathtt{pwrite(p,}\ r\mathtt{)}, i'\rangle}$$

$$\frac{i \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}^{\mathsf{S}} : \langle l, v, r, i'\rangle \qquad i' \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s}_1 : \langle l_1, r_1, i_1\rangle \qquad i' \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s}_2 : \langle l_2, r_2, i_2\rangle}{\begin{array}{l} i \vdash^{\mathrm{s}}_{\mathrm{d}} \mathtt{if}\ \mathtt{(}\hat{e}^{\mathsf{S}}\mathtt{)}\ \hat{s}_1\ \mathtt{else}\ \hat{s}_2 : \\ \quad \langle \mathtt{\{}l\mathtt{;if}\ \mathtt{(}v\mathtt{)}\ \mathtt{\{}l_1\mathtt{;cache = cache+}i_1\mathtt{\}}\ \mathtt{else}\ \mathtt{\{}l_2\mathtt{;cache = cache+}i_2\mathtt{\}\}}, \\ \quad\ \mathtt{if}\ \mathtt{(}r\mathtt{)}\ \mathtt{\{}r_1\mathtt{;cache = cache+}i_1\mathtt{\}}\ \mathtt{else}\ \mathtt{\{}r_2\mathtt{;cache = cache+}i_2\mathtt{\}}, \\ \quad\ 0\rangle \end{array}}$$

$$\frac{i \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s}_1^{b_1} : \langle l_1, r_1, i_1\rangle \qquad \ldots \qquad i_{n-1} \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s}_n^{b_n} : \langle l_n, r_n, i_n\rangle}{i \vdash^{\mathrm{s}}_{\mathrm{d}} \mathtt{\{}\hat{s}_1^{b_1}\mathtt{;}\ldots\mathtt{;}\hat{s}_n^{b_n}\mathtt{\}} : \langle \mathtt{\{}l_1\mathtt{;}\ldots\mathtt{;}l_n\mathtt{\}}, \mathtt{\{}r_1\mathtt{;}\ldots\mathtt{;}r_n\mathtt{\}}, i_n\rangle}$$

Expressions:

$$i \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}^{\mathsf{S}} : \langle \mathtt{*(cache+}i\mathtt{)} = e, \mathtt{*(cache+}i\mathtt{)}, \mathtt{*(cache+}i\mathtt{)}, i+1\rangle \qquad\qquad i \vdash^{\mathrm{e}}_{\mathrm{d}} \mathtt{x}^{\mathsf{D}} : \langle \mathtt{\{\}}, 0, \mathtt{x}, i\rangle$$

$$i \vdash^{\mathrm{e}}_{\mathrm{d}} \mathtt{pread(p}^{\mathsf{S}}\mathtt{)}^{\mathsf{D}} : \langle \mathtt{\{\}}, 0, \mathtt{*p}, i\rangle \qquad\qquad i \vdash^{\mathrm{e}}_{\mathrm{d}} \mathtt{pread(p}^{\mathsf{D}}\mathtt{)}^{\mathsf{D}} : \langle \mathtt{\{\}}, 0, \mathtt{pread(p)}, i\rangle$$

$$\frac{i \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}_1^{b_1} : \langle l_1, v_1, r_1, i_1\rangle \qquad i_1 \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}_2^{b_2} : \langle l_2, v_2, r_2, i_2\rangle}{i \vdash^{\mathrm{e}}_{\mathrm{d}} (\hat{e}_1^{b_1}\ op\ \hat{e}_2^{b_2})^{\mathsf{D}} : \langle \mathtt{\{}l_1\mathtt{;}l_2\mathtt{\}}, 0, r_1\ op\ r_2, i_2\rangle}$$

Fig. 2. Data specialization of statements and expressions (excerpts)

The generated program first initializes `cache` to the beginning of the cache and the static persistent variables to cache entries. Next the loader l is executed. Finally, the value of `cache` is reset, and the reader r is executed.

4 Correctness

We now relate the semantics of the result of data specialization to the semantics of the source program. We begin with the semantics of the target language, which is a superset of the source language. The semantics depends on a store σ mapping locations to values. For conciseness, we implicitly associate each variable x with the store location $\tilde{\mathsf{x}}$. The semantics of a statement s is specified by $\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} s : \sigma'$, for stores σ and σ'. The semantics of an expression e is specified by $\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e : v$, where σ is a store and v is a value. We only describe the semantics of the constructs manipulating persistent variables; the other constructs are standard, and are deferred to Appendix B.

The semantics must ensure that every reference to a persistent variable using `pread` sees the final value to which the variable is assigned using `pwrite`. We use two distinct store locations $\tilde{\mathtt{p}}^{\mathrm{in}}$ and $\tilde{\mathtt{p}}^{\mathrm{out}}$ to represent each persistent variable

p. The location $\tilde{\mathsf{p}}^{\mathrm{in}}$ holds the final value of p, while the location $\tilde{\mathsf{p}}^{\mathrm{out}}$ records updates to p. Thus, the semantics of `pread` and `pwrite` are as follows, where *undefined* is some value distinct from the value of any expression:

$$\frac{v \neq \mathit{undefined}}{\sigma[\tilde{\mathsf{p}}^{\mathrm{in}} \mapsto v] \vdash^{\mathrm{e}}_{\mathrm{s}} \texttt{pread(p)} : v} \qquad \frac{\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e : v}{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} \texttt{pwrite(p,}\ e\texttt{)} : \sigma[\tilde{\mathsf{p}}^{\mathrm{out}} \mapsto v]}$$

The values stored in $\tilde{\mathsf{p}}^{\mathrm{in}}$ and $\tilde{\mathsf{p}}^{\mathrm{out}}$ are connected by the semantics of a complete program, specified as follows:

Definition 1. *Let p be a program $d_1 \ldots d_n\, s$ declaring the variables $\mathsf{x}_1, \ldots, \mathsf{x}_y$ and the persistent variables $\mathsf{p}_1, \ldots, \mathsf{p}_q$. Let σ_0 be a state mapping the $\tilde{\mathsf{x}}_i$ to the initial values of the x_i, and the $\tilde{\mathsf{p}}_j^{\mathrm{out}}$ to "undefined". Then, the meaning of p, $[\![p]\!]$, is the set of stores σ, binding only the $\tilde{\mathsf{x}}_i$, such that for some values $v_1, \ldots, v_q$*

$$\sigma_0 \cup \{\tilde{\mathsf{p}}_j^{\mathrm{in}} \mapsto v_j \mid 1 \leq j \leq q\} \vdash^{\mathrm{s}}_{\mathrm{s}} s : \sigma \cup \{\tilde{\mathsf{p}}_j^{\mathrm{in}}, \tilde{\mathsf{p}}_j^{\mathrm{out}} \mapsto v_j \mid 1 \leq j \leq q\}$$

This definition uses the value *undefined* to ensure that the meaning of a program represents computations in which the value of a persistent variable is only read when it is also defined.

We now show that data specialization preserves the semantics. Data specialization separates a program into the loader, which only manipulates static variables, and the reader, which only manipulates dynamic variables. To relate the semantics of the loader and reader to the semantics of the source program, we first define the operators stat and dyn, which separate the input store into static and dynamic components:

Definition 2. *Let σ be a store and Γ be a binding-time environment. Let $\bar{\mathsf{p}}^{\mathrm{out}}$ and $\bar{\mathsf{p}}^{\mathrm{in}}$ be locations that are unique for each persistent variable p, but that may be identical to each other. Then,*

1. $\mathsf{stat}(\sigma, \Gamma) = \{\tilde{\mathsf{x}} \mapsto \sigma(\tilde{\mathsf{x}}) \mid \Gamma(\mathsf{x}) = \mathsf{S}\} \cup \{\tilde{\mathsf{p}} \mapsto \bar{\mathsf{p}}^{\mathrm{out}}, \bar{\mathsf{p}}^{\mathrm{out}} \mapsto \sigma(\tilde{\mathsf{p}}^{\mathrm{out}}) \mid \Gamma(\mathsf{p}) = \mathsf{S}\}$
2. $\mathsf{dyn}(\sigma, \Gamma) = \{\tilde{\mathsf{x}} \mapsto \sigma(\tilde{\mathsf{x}}) \mid \Gamma(\mathsf{x}) = \mathsf{D}\} \cup \{\tilde{\mathsf{p}} \mapsto \bar{\mathsf{p}}^{\mathrm{in}}, \bar{\mathsf{p}}^{\mathrm{in}} \mapsto \sigma(\tilde{\mathsf{p}}^{\mathrm{in}}) \mid \Gamma(\mathsf{p}) = \mathsf{S}\} \cup$
   $\{\tilde{\mathsf{p}}^{\mathrm{in}} \mapsto \sigma(\tilde{\mathsf{p}}^{\mathrm{in}}), \tilde{\mathsf{p}}^{\mathrm{out}} \mapsto \sigma(\tilde{\mathsf{p}}^{\mathrm{out}}) \mid \Gamma(\mathsf{p}) = \mathsf{D}\}$

To relate stores $\mathsf{stat}(\sigma, \Gamma)$ and $\mathsf{dyn}(\sigma, \Gamma)$ back to σ, we must eliminate the intermediate locations $\bar{\mathsf{p}}^{\mathrm{in}}$ and $\bar{\mathsf{p}}^{\mathrm{out}}$. We thus define the operator $\uplus$:

Definition 3. *For binding-time environment Γ and stores α and β,*

$$\begin{aligned}\alpha \uplus_\Gamma \beta = &\{\tilde{\mathsf{x}} \mapsto \alpha(\tilde{\mathsf{x}}) \mid \Gamma(\mathsf{x}) = \mathsf{S}\} \cup \{\tilde{\mathsf{x}} \mapsto \beta(\tilde{\mathsf{x}}) \mid \Gamma(\mathsf{x}) = \mathsf{D}\}\\ &\cup \{\tilde{\mathsf{p}}^{\mathrm{in}} \mapsto \beta(\bar{\mathsf{p}}^{\mathrm{in}}), \tilde{\mathsf{p}}^{\mathrm{out}} \mapsto \alpha(\bar{\mathsf{p}}^{\mathrm{out}}) \mid \Gamma(\mathsf{p}) = \mathsf{S}\}\\ &\cup \{\tilde{\mathsf{p}}^{\mathrm{in}} \mapsto \beta(\tilde{\mathsf{p}}^{\mathrm{in}}), \tilde{\mathsf{p}}^{\mathrm{out}} \mapsto \beta(\tilde{\mathsf{p}}^{\mathrm{out}}) \mid \Gamma(\mathsf{p}) = \mathsf{D}\}\end{aligned}$$

The operators stat, dyn, and $\uplus$ are related by the following lemma:

Lemma 1. $\mathsf{stat}(\sigma, \Gamma) \uplus_\Gamma \mathsf{dyn}(\sigma, \Gamma) = \sigma$

The loader and reader also use some store locations not present in the store used by the source programs, namely the cache pointer, the local `tmp` variables, and the cache. For conciseness, we specify the store with respect to which the loader and reader are executed as a sequence $\alpha, \mathsf{c}, \tau, \xi$ of

1. The bindings (α) associated with the source variables.
2. The cache pointer (c).
3. A stack (τ) representing the values of the locally declared `tmp` variables.
4. The cache (ξ).

The following theorem shows that given the loader and reader associated with a source statement, execution of the loader followed by resetting of the cache pointer followed by execution of the reader has the same effect on the values of the source variables as execution of the source statement.

Theorem 1. *For any statement s and store σ, if $\mathit{\Gamma}, b_c \vdash_{\mathrm{s}} s : \hat{s}$ and $i \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s} : \langle l, r, i' \rangle$, and if for some c, τ, and ξ, there are α, β, c', τ', and ξ' such that*

1. $\mathsf{stat}(\sigma, \mathit{\Gamma}), \mathsf{c}, \tau, \xi \vdash^{\mathrm{s}}_{\mathrm{s}} l : \alpha, \mathsf{c}', \tau', \xi'$.
2. $\mathsf{dyn}(\sigma, \mathit{\Gamma}), \mathsf{c}, \tau', \xi' \vdash^{\mathrm{s}}_{\mathrm{s}} r : \beta, \mathsf{c}', \tau', \xi'$.

Then, $\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} s : \alpha \uplus \beta$.

Because of speculative evaluation of terms under dynamic control (see Appendic A), termination of the source statement does not necessarily imply termination of the loader. Thus, we relate the semantics of the source program to that of the loader and reader using the following theorem, which includes the hypothesis that the loader terminates.

Theorem 2. *For any statement s and store σ, if $\mathit{\Gamma}, b_c \vdash_{\mathrm{s}} s : \hat{s}$ and $i \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s} : \langle l, r, i' \rangle$, and there are some σ', c, τ, and ξ such that*

1. $\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} s : \sigma'$
2. *For some store σ'', $\mathsf{stat}(\sigma, \mathit{\Gamma}), \mathsf{c}, \tau, \xi \vdash^{\mathrm{s}}_{\mathrm{s}} l : \sigma''$*

Then, there are some α, β, c', τ', and ξ', such that

1. $\mathsf{stat}(\sigma, \mathit{\Gamma}), \mathsf{c}, \tau, \xi \vdash^{\mathrm{s}}_{\mathrm{s}} l : \alpha, \mathsf{c}', \tau', \xi'$.
2. $\mathsf{dyn}(\sigma, \mathit{\Gamma}), \mathsf{c}, \tau', \xi' \vdash^{\mathrm{s}}_{\mathrm{s}} r : \beta, \mathsf{c}', \tau', \xi'$.
3. $\sigma' = \alpha \uplus \beta$

Both theorems are proved by induction on the height of the derivation of $\mathit{\Gamma}, b_c \vdash_{\mathrm{s}} s : \hat{s}$. These theorems imply the following, which shows that the store resulting from execution of the source statement and the store resulting from execution of the result of data specialization agree on the variables x, which determine the semantics of a program as specified by Definition 1.

Corollary 1. *For any program $p \equiv d_1 \ldots d_n\, s$, binding-time environment $\mathit{\Gamma}$, and store σ, if $\mathsf{ds}[\![p]\!]_{\mathit{\Gamma}} \equiv d'_1 \ldots d'_m\, s_{ds}$, $\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} s : \sigma'$, and $\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} s_{ds} : \sigma'_{ds}$, then for all variables x of p, $\sigma'(\mathsf{x}) = \sigma'_{ds}(\mathsf{x})$.*

The corollary holds because s_{ds} initially sets up a store that is compatible with the store assumed by the above theorems and resets the cache pointer between the execution of the loader and the reader.

5 Circularity and attribute grammars

The efficient implementation of circular specifications has been studied extensively in the attribute-grammar community [13, 18–21]. As is the case for offline partial evaluation, these approaches begin with a dependency analysis, the results of which then guide code generation. Nevertheless, the partial-evaluation and attribute-grammar-based approaches differ in their starting point and in the quantity of information collected by the analyses. Our approach also differs from attribute-grammar-based approaches in that our source language is imperative.

The starting point of an attribute-grammar-based approach is an attribute grammar describing the input of the program to be generated. An analysis determines the dependencies between the attributes, and uses this information to construct a series of schedules, known as a *visit sequences*, of the attribute computations such that each computation depends only on previously-computed attributes. Each element of a visit sequence can be implemented as a function whose argument is the component of the input for which the corresponding attribute is to be computed. The resulting implementation amounts to a series of recursive traversals of the input structure [13]. Specific techniques have been devised to transmit intermediate values ("bindings" [18]) that are not part of the input from one phase to the next, and to create specialized input structures ("visit trees" [19, 21]) that eliminate the need to maintain portions of the input that are not needed by subsequent phases.

In contrast, the starting point of data specialization is the program itself; data specialization is independent of the structure of the input data. The construction of both glue and visit trees is subsumed by the basic mechanism of data specialization: the caching of the value of every static expression that occurs in a dynamic context. The cache can, nevertheless, be viewed as implementing a specialized visit tree. The cached values of static conditional tests correspond to sum-type tags, while the values cached for the chosen branch of a static conditional correspond to the components of the sum element. In our implementation, this tree structure is flattened; pointers from one cache entry to another are only introduced to implement speculative evaluation of dynamic control constructs (See Appendix A).

The binding-time analysis used by data specialization is significantly less informative than the analysis used in the implementation of attribute grammars. We have seen that binding-time analysis simply classifies each expression as static or dynamic, according to the classification of the terms that it depends on. This strategy implies that a persistent variable that depends on the value of another instance of itself is considered dynamic. The attribute-grammar analysis collects complete information about the relationships between attributes. This extra precision allows fewer dependencies to be considered recursive. The impact of replacing the binding-time analysis of data specialization by the more informative attribute-grammar analysis is a promising area for future research.

Finally, we have presented an implementation of circularity in the context of an imperative language, whereas attribute grammars use a declarative notation, with some similarity to a lazy functional language [10]. Because imperative languages are flow sensitive, we have to indicate explicitly which variables should be considered persistent, and thus take on the final value to which they are assigned, rather than the most recent value. In contrast, in a flow-insensitive language, such as a declarative or functional language, a variable has only one value, which is its final value. The addition of circularity to such a language simply allows the value to be specified after the point of reference, rather than requiring that it be specified before, and no keyword is required.

6 Examples

We now present some examples: the translation of the imperative definition of repmin, a use of our approach in the implementation of run-time specialization in Tempo, and two examples from the literature on circular programs.

6.1 Repmin

Figure 3 shows the result of applying data specialization to the imperative implementation of repmin presented in Section 1.[2] The loader, comprised of `rm_ldr` and `repmin_ldr`, accumulates the minimum value at any `Tip`. Once the complete tree has been analyzed, this value is stored in the cache location assigned to the persistent variable `minval`. The reader, comprised of `rm_rdr` and `repmin_rdr`, uses this value to reconstruct the tree.

In the implementation of Figure 3, calls to the primitives `mkTip` and `mkFork` are considered to be dynamic, and thus placed in the reader. It is also possible to consider these calls to be static, in which case the output structure is built in the loader. Following this strategy, the reader recursively visits the tips, instantiating the value of each to be the minimum value. When part of the output structure does not depend on the values of persistent variables, this strategy implies that the binding-time analysis considers its construction to be completely static, which can reduce the amount of data stored in the cache.

6.2 Inlining

This work was motivated by the problem of optimizing run-time specialization in Tempo [15]. An important optimization is the inlining of specialized functions. Inlining is performed during the execution of a dedicated specializer (*generating extension* [11]) written in C, and is thus most naturally implemented in C as well. We have found that the use of persistent variables facilitates the implementation of various inlining strategies, by requiring only local changes that do not affect the overall implementation of run-time specialization.

To achieve good performance, the size of a specialized function should not exceed the size of the instruction cache or the distance expressible by a relative

[2] The code produced by Tempo has been slightly modified for readability. Among these simplifications, we exploit the fact that only integers are stored in the cache to eliminate casts to and from a generic cache type.

```
void rm_ldr(Tree *t, int *cache) {
 Ans a;
 int *minval_ptr = cache++;
 cache =
  repmin_ldr(t, &a, cache);
 *minval_ptr = a.mn;
}

int *repmin_ldr(Tree *t, Ans *a,
                int *cache) {
 Ans a1, a2;
 *cache = (t->type == Fork);
 if (*cache++) {
  cache =
   repmin_ldr(t->left,  &a1, cache);
  cache =
   repmin_ldr(t->right, &a2, cache);
  a->mn = min(a1.mn, a2.mn);
 }
 else a->mn = t->tipval;
 return cache;
}

Tree *rm(Tree *t) {
  int *cache = mkCache();
  rm_ldr(t, cache);
  return rm_rdr(cache);
}
```

```
Tree *rm_rdr(int *cache) {
 Ans a;
 int minval = *cache++;
 cache =
  repmin_rdr(minval, &a, cache);
 return a.tree;
}

int *repmin_rdr(int m, Ans *a,
                int *cache) {
 Ans a1, a2;
 if (*cache++) {
  cache =
   repmin_rdr(m, &a1, cache);
  cache =
   repmin_rdr(m, &a2, cache);
  a->tree =
   mkFork(a1.tree, a2.tree);
 }
 else a->tree = mkTip(m);
 return cache;
}
```

Fig. 3. Data specialization of repmin

branch instruction. One approach is to constrain inlining based on the number of instructions already generated for the current function. A more precise approach is to constrain inlining based on the size of the complete specialized function. To implement these strategies, we use data specialization and persistent variables to separate the implementation into a pass that analyzes the size, followed by a pass that performs the inlining and code generation.

Inlining based on current function size: The heart of the implementation is the function `do_call`, shown with binding-time annotations in Figure 4. The arguments to `do_call` are the number of instructions already generated for the caller (`caller_size`), the name of the callee (`callee`), and the buffer into which code for the caller is generated (`caller_output`). The treatment of a call proceeds in three steps. First, `callee_output` is initialized to the address at which to generate code for the specialized callee. If the specialized callee is to be inlined, `callee_output` is set to the current position in the caller's buffer, as indicated by `caller_output`. Otherwise, `mkFun` is used to allocate a new buffer that is

```
extern Code *the_program[];
extern int threshold;

int do_call(int caller_size, int callee, Code *caller_output) {
  int inlined, callee_size;
  persistent int inlinedp, callee_sizep;
  Code *callee_output;

  /* select the output buffer based on whether the call is inlined */
  if (pread(inlinedp))
    callee_output = caller_output;
  else
    callee_output = mkFun(pread(callee_sizep));

  /* specialize the callee */
  callee_size = spec(the_program[callee], callee_output);

  /* initializations based on whether the call is inlined */
  inlined = (callee_size + caller_size <= threshold);
  pwrite(inlinedp,inlined);
  if (inlined)
    /* return the number of instructions added to the caller */
    return callee_size;
  else {
    /* end the callee */
    *(callee_output + callee_size) = RETURN;
    /* record the callee's size */
    pwrite(callee_sizep,callee_size+1);
    /* add a call instruction to the caller */
    *output = mkCall(get_name(callee_output));
    /* return the number of instructions added to the caller */
    return 1;
  }
}
```

Fig. 4. The `do_call` function used in the implementation of inlining based on current function size. Dynamic constructs are underlined.

the size of the specialized callee. Because the decision of whether to inline and the size of the specialized callee are not known until after specialization of the callee, this information is implemented using the persistent variables `inlinedp` and `callee_sizep`, respectively. Next, the callee is specialized by applying the function `spec` to the callee's source code and the selected output buffer. Specialization emits code in the output buffer and returns the size of the specialized function. Finally, the number of instructions already generated for the caller is combined with the size of the specialized callee to determine whether the callee should be inlined. The persistent variables are then initialized accordingly, and other initializations are performed as indicated by the comments in Figure 4. The return value is the number of instructions added to the caller.

The loader produced by data specialization computes the size of each generated function and determines whether it should be inlined. The reader then uses this information to allocate the output buffer and perform code generation.

Controlling inlining based on maximum possible function size: The previous approach takes into account only the number of instructions generated for the caller so far. A more accurate approach is to consider the caller's total size. For this purpose we add a new persistent variable `local_sizep` recording the total size of the current function before inlining.

The use of the dynamic value of the persistent variable `local_sizep` to determine whether to inline implies that the value of the persistent variable `inlinedp` depends on dynamic information. Thus, we must iterate data specialization, producing a three-phase implementation. The first phase calculates the number of instructions generated for each source function, if no inlining is performed. The second phase decides whether to inline each call, based on the sum of the number of instructions in the specialized caller and the number of instructions added by inlining all selected calls. The third phase performs the actual code generation.

6.3 Other circular programs

We now consider several examples from the literature on attribute-grammar-based and lazy implementations of circular programs. These examples illustrate the limitations of an approach based on binding-time analysis.

The Block language has been used by Saraiva *et al.* to illustrate numerous strategies for generating efficient implementations of attribute grammars [19, 20, 22]. Block is a block-structured language in which the scope of a variable declaration is any nested block, including nested blocks to the left of the declaration. The language thus generalizes the common use of forward references to top-level functions, extending this facility to local variables. The scoping rule is illustrated by the following program, in which braces delimit a nested block, "`int x`" is a declaration of `x`, and "`x`" is a use of `x`:

```
{ { x } int x }
```

We focus on one of the compilation problems that has been studied for Block, that of translating Block code to a stack-based language [22]. Circularity arises when the compiler needs to determine the stack offset of a variable that occurs before its declaration has been processed.

Saraiva *et al.* present two implementations of such a compiler: a lazy implementation and a strict implementation generated from an attribute grammar [22]. Both implementations represent the environment as two variables, which we refer to as the *local* environment and the *complete* environment. The local environment contains all of the declarations for the enclosing blocks, but only the variables declared to the left of the current position in the current block. As new declarations are encountered, they are recorded in this environment. The complete environment contains all of the declarations that should be visible at the current point, including those that occur to the right of the current position. This environment is used for code generation. In the compilation of a block, the environments are connected using a circular reference: the input value of the complete environment is the local environment that results from processing the block.

This implementation can be directly translated into our language by using a persistent variable to implement the complete environment. If we follow this strategy, however, our approach is unable to eliminate the circularity. Because the initial value of the local environment is the dynamic value of the persistent variable representing the complete environment of the enclosing block, the local environment is always dynamic, and cannot be used to initialize the complete environment of the nested block to a static value. The strict implementation of Saraiva *et al.* resolves this dependency by a strategy analogous to calling the loader for the treatment of a block from the reader, rather than from the loader of the context. Within the reader, the complete environment for the surrounding context has been determined. It is, however, not clear how to infer the need for this implementation strategy using binding-time analysis.

By slightly reorganizing the structure of the environment, we can implement the Block compiler using persistent variables and data specialization. The local environment of a nested block extends the enclosing block's complete environment, but does not otherwise depend on its contents. We thus replace the flat environment generated by Saraiva *et al.*'s implementation by an environment constructed of frames, such that an empty, and thus static, frame is allocated on entry to each block. The loader adds the declarations made by the block to this frame, which is then stored in a persistent variable at the end of the block. When the reader enters the block, it extends the complete environment with this new frame, producing an environment suitable for code generation. This program structure is also used in the second inlining example. The reorganization corresponds roughly to reformulating a computation $f(d)$ into $d \oplus f(s)$, where s is some static initial value and $\oplus$ is some operation, thus permitting the computation of f to be considered static.

A related example is Karczmarczuk's use of circularity to concisely implement complex mathematical operations in a lazy functional language [12]. Like the compiler of the BLOCK language, Karczmarczuk's implementation has the property that a circular value from an enclosing computation is used in performing a subcomputation. Here, however, the value from the enclosing computation is used eagerly, and it is not clear how to perform a rewriting of the form of the conversion of $f(d)$ into $d \oplus f(s)$.

7 Related work

The most closely related work is the automatic efficient implementation of an attribute-grammar specification, which has been discussed in Section 5. Here, we review the history of data specialization and of multiple levels of specialization.

Data specialization: Automatic data specialization was initially developed by Barzdins and Bulyonkov [2], and described and extended by Malmkjær [16]. These approaches are more complex than ours. In particular, they use memoization in the construction of the data specialization cache. Knoblock and Ruf implement data specialization for a subset of C and investigate its use in an interactive graphics application [14]. Chirokoff *et al.* compare the benefits of program and data specialization, and propose combining these techniques [5]. Our implementation of data specialization in Tempo builds on that of Chirokoff.

Incremental specialization: We have proposed to iterate binding-time analysis and data specialization to resolve dependencies among a hierarchy of persistent variables. Marlet, Consel, and Boinot similarly iterate the specialization process to achieve incremental run-time program specialization [17]. Alternatively, Glück and Jørgensen define a binding-time analysis and program specializer that treat multiple levels at once [7]. Their analysis should be applicable to our approach.

8 Conclusions and future work

In this paper, we have shown how circular programs can be implemented using a minor extension of standard partial evaluation techniques. Previously developed techniques to generate optimized implementations of circular specifications are naturally achieved by the basic strategy of caching the values of static expressions that occur in a dynamic context. We have found the use of persistent variables crucial in experimenting with a variety of optimization strategies for run-time specialization in Tempo. Because the introduced code is localized, and a staged program is generated automatically, variants can be implemented robustly and rapidly.

In future work, we plan to allow persistent variables as first-class values. Given the set of analyses already performed by Tempo [8], this extension should be straightforward. We also plan to investigate whether the information collected by the analysis used for attribute grammars can be useful in the context of partial evaluation. We hope that the work presented here will lead to further exchange of techniques between the attribute-grammar and partial-evaluation communities.

Acknowledgements: This work has greatly benefitted from the encouragement and feedback of many people, including Olivier Danvy, Andrzej Filinski, Bernd Grobauer, Gilles Muller, Renaud Marlet, Ulrik Schultz, and the PADO-II reviewers.

References

1. A.W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998. pages 131–132, 138–139.
2. G.J. Barzdins and M.A. Bulyonkov. Mixed computation and translation: Linearisation and decomposition of compilers. Preprint 791 from Computing Centre of Sibirian division of USSR Academy of Sciences, p.32, Novosibirsk, 1988.
3. R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984/85.
4. A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.
5. S. Chirokoff, C. Consel, and R. Marlet. Combining program and data specialization. *Higher-Order and Symbolic Computation*, 12(4):309–335, December 1999.
6. C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.
7. R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10:113–158, 1997.
8. L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoretical Computer Science*, 248(1–2):3–27, 2000.
9. L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, September 1997. Springer-Verlag.
10. T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn, editor, *1987 Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173, Portland, OR, September 1987. Springer-Verlag.
11. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, June 1993.
12. J. Karczmarczuk. Calcul des adjoints et programmation paresseuse. In *Journées Francophones des Langages Applicatifs (JFLA'2001)*, Metabief, France, January 2001.
13. U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980.
14. T.B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 215–225, Philadelphia, PA, May 1996. ACM SIGPLAN Notices, 31(5). Also TR MSR-TR-96-04, Microsoft Research, February 1996.
15. J. Lawall and G. Muller. Faster run-time specialized code using data specialization. Research Report 3833, INRIA, Rennes, France, December 1999.

16. K. Malmkjær. Program and data specialization: Principles, applications, and self-application. Master's thesis, DIKU University of Copenhagen, August 1989.
17. R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 281–292, Atlanta, GA, May 1999.
18. M. Pennings. *Generating incremental attribute evaluators*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, November 1994.
19. J. Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999.
20. J. Saraiva, D. Swierstra, and M. Kuiper. Strictification of computations on trees. In *3th Latin-American Conference on Functional Programming (CLaPF'99)*, March 1999.
21. J. Saraiva, D. Swierstra, and M. Kuiper. Functional incremental attribute evaluation. In D.A. Watt, editor, *9th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 279–294, Berlin, Germany, March 2000. Springer-Verlag.
22. J. Saraiva, D. Swierstra, M. Kuiper, and M. Pennings. Strictification of lazy functions. Technical Report UU-CS 1996-51, Utrecht University, 1996.
23. E. Sumii and N. Kobayashi. Online-and-offline partial evaluation: A mixed approach (extended abstract). In *2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, pages 12–21. ACM Press, 2000. Also available as *SIGPLAN Notices* 34(11).

A Data specialization of branching statements

The data specialization rules for conditionals and while loops are shown in Figure 5. The principal problem here is to maintain the cache pointer. For static control constructs, all possible control paths must set the cache pointer such that a single constant offset i can be used after the control construct. The speculative evaluation performed for dynamic control constructs implies that cache entries are initialized in the loader that correspond to code that is not executed in the reader. Thus, the cache itself has to record which cache entries to skip, according to the control path chosen in the reader. While speculative evaluation is not essential, it has been found useful in practice [11].

The specialization rules in Figure 5 create a cache entry for the value of the test of each static conditional and for the value of the test performed on each static while loop iteration. A more efficient approach is to collapse nested static conditionals into a `switch` statement and to replace the recording of the values of while loop tests by the recording of the number of loop iterations. Both optimizations have been implemented in Tempo.

B Semantics

The complete semantics of statements and expressions is shown in Figure 6.

$$\frac{i \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}^{\mathsf{S}} : \langle l, v, r, i' \rangle \qquad i' \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s}_1 : \langle l_1, r_1, i_1 \rangle \qquad i' \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s}_2 : \langle l_2, r_2, i_2 \rangle}{\begin{array}{l} i \vdash^{\mathrm{s}}_{\mathrm{d}} \texttt{if (}\hat{e}^{\mathsf{S}}\texttt{)}\ \hat{s}_1\ \texttt{else}\ \hat{s}_2 : \\ \quad \langle \texttt{\{}l\texttt{;if (}v\texttt{) \{}l_1\texttt{;cache = cache+}i_1\texttt{\} else \{}l_2\texttt{;cache = cache+}i_2\texttt{\}\}}, \\ \quad\ \texttt{if (}r\texttt{) \{}r_1\texttt{;cache = cache+}i_1\texttt{\} else \{}r_2\texttt{;cache = cache+}i_2\texttt{\}}, \\ \quad\ 0 \rangle \end{array}}$$

$$\frac{i \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}^{\mathsf{D}} : \langle l, v, r, i' \rangle \qquad i' + 1 \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s}_1 : \langle l_1, r_1, i_1 \rangle \qquad i_1 + 1 \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s}_2 : \langle l_2, r_2, i_2 \rangle}{\begin{array}{l} i \vdash^{\mathrm{s}}_{\mathrm{d}} \texttt{if(}\hat{e}^{\mathsf{D}}\texttt{)}\ \hat{s}_1\ \texttt{else}\ \hat{s}_2 : \\ \langle \texttt{\{Cache tmp;}\ l\texttt{;tmp = cache+}i'\texttt{;}l_1\texttt{;*tmp = cache;tmp = cache+}i_1\texttt{;}l_2\texttt{;*tmp = cache\}}, \\ \ \texttt{if (}r\texttt{) \{}r_1\texttt{;cache = *(cache+}i_1\texttt{)\} else \{cache = *(cache+}i'\texttt{)}r_2\texttt{\}}, \\ \ i_2 \rangle \end{array}}$$

$$\frac{i \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}^{\mathsf{S}} : \langle l, v, r, i' \rangle \qquad i' \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s} : \langle l_s, r_s, i_s \rangle}{\begin{array}{l} i \vdash^{\mathrm{s}}_{\mathrm{d}} \texttt{while (}\hat{e}^{\mathsf{S}}\texttt{)}\ \hat{s} : \langle \texttt{\{}l\texttt{;while (}v\texttt{) \{}l_s\texttt{;cache = cache} + i_s - i\texttt{;}l\texttt{\}\}}, \\ \qquad\qquad\qquad\qquad \texttt{while (}r\texttt{) \{}r_s\texttt{;cache = cache} + i_s - i\texttt{\}}, \\ \qquad\qquad\qquad\qquad i_s \rangle \end{array}}$$

$$\frac{i \vdash^{\mathrm{e}}_{\mathrm{d}} \hat{e}^{\mathsf{D}} : \langle l, v, r, i' \rangle \qquad i' + 1 \vdash^{\mathrm{s}}_{\mathrm{d}} \hat{s} : \langle l_s, r_s, i_s \rangle}{\begin{array}{l} i \vdash^{\mathrm{s}}_{\mathrm{d}} \texttt{while (}\hat{e}^{\mathsf{D}}\texttt{)}\ \hat{s} : \\ \quad \langle \texttt{\{Cache tmp;}l\texttt{;tmp = cache} + i'\texttt{;}l_s\texttt{;*tmp = cache\}}, \\ \quad\ \texttt{\{Cache tmp;tmp = cache;while (}r\texttt{) \{}r_s\texttt{;cache = tmp\};cache = *(cache} + i'\texttt{)\}}, \\ \quad\ i_s \rangle \end{array}}$$

Fig. 5. Data specialization of branching statements

Statements:

$$\frac{\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e : v}{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} \mathsf{x} = e : \sigma[\tilde{\mathsf{x}} \mapsto v]} \qquad \frac{\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e_1 : \ell \qquad \sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e_2 : v}{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} *e_1 = e_2 : \sigma[\ell \mapsto v]} \qquad \frac{\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e : v}{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} \texttt{pwrite(p,}\ e\texttt{)} : \sigma[\tilde{\mathsf{p}}^{\mathrm{out}} \mapsto v]}$$

$$\frac{\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e : 1 \qquad \sigma \vdash^{\mathrm{s}}_{\mathrm{s}} s_1 : \sigma'}{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 : \sigma'} \qquad \frac{\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e : 0 \qquad \sigma \vdash^{\mathrm{s}}_{\mathrm{s}} s_2 : \sigma'}{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 : \sigma'}$$

$$\frac{\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e : 1 \qquad \sigma \vdash^{\mathrm{s}}_{\mathrm{s}} s : \sigma' \qquad \sigma' \vdash^{\mathrm{s}}_{\mathrm{s}} \texttt{while (}e\texttt{)}\ s : \sigma''}{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} \texttt{while (}e\texttt{)}\ s : \sigma''} \qquad \frac{\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e : 0}{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} \texttt{while (}e\texttt{)}\ s : \sigma}$$

$$\frac{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} s_1 : \sigma_1 \qquad \ldots \qquad \sigma_{n-1} \vdash^{\mathrm{s}}_{\mathrm{s}} s_n : \sigma_n}{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} \texttt{\{}s_1\texttt{;} \ldots \texttt{;}s_n\texttt{\}} : \sigma_n} \qquad \frac{\sigma[\tilde{\mathsf{x}} \mapsto \mathit{undefined}] \vdash^{\mathrm{s}}_{\mathrm{s}} s : \sigma'[\tilde{\mathsf{x}} \mapsto v]}{\sigma \vdash^{\mathrm{s}}_{\mathrm{s}} \texttt{\{Cache x;}s\texttt{\}} : \sigma'}$$

Expressions:

$$\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} c : c \qquad \sigma[\tilde{\mathsf{x}} \mapsto v] \vdash^{\mathrm{e}}_{\mathrm{s}} \mathsf{x} : v \qquad \frac{\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e_1 : v_1 \qquad \sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e_2 : v_2}{\sigma \vdash^{\mathrm{e}}_{\mathrm{s}} e_1\ \mathit{op}\, e_2 : v_1\ \mathit{op}\, v_2}$$

$$\frac{\sigma[\ell \mapsto v] \vdash^{\mathrm{e}}_{\mathrm{s}} e : \ell}{\sigma[\ell \mapsto v] \vdash^{\mathrm{e}}_{\mathrm{s}} *e : v} \qquad \frac{v \neq \mathit{undefined}}{\sigma[\tilde{\mathsf{p}}^{\mathrm{in}} \mapsto v] \vdash^{\mathrm{e}}_{\mathrm{s}} \texttt{pread(p)} : v}$$

Fig. 6. Semantics of statements and expressions