Generalization in Hierarchies of Online Program Specialization Systems
(Extended Abstract) *

Robert Glück[†] John Hatcliff[‡] Jesper Jørgensen[§]

Abstract

In recent work, we proposed a simple functional language S-graph-n to study metaprogramming aspects of self-applicable online program specialization. The primitives of the language provide support for multiple encodings of programs. An important component of online program specialization is the termination strategy. In this paper we show that such a representation has the great advantage of simplifying generalization of multiply encoded data. We extend two basic methods to multiply encoded data: most specific generalization and the homeomorphic embedding relation. Examples illustrate their working in hierarchies of programs.

1 Introduction

Multiple levels of metaprograms have become an issue for several reasons, on the one hand they are a natural extension of metaprogramming approaches, on the other hand they have been used to great advantage for program generation. Multi-level metaprogramming languages that support this style of metaprogramming have been advocated, *e.g.* [3, 16]. They are a key ingredient in the design and implementation of *generative software*, *e.g.* [7].

In recent work [10], we proposed a simple language S-graph-n to study meta-programming aspects of self-applicable online program specialization. The primitives of the language provide support for multiple encodings of programs and the use of metavariables to track unknown values. They were motivated by the need for space and time efficient representation and manipulation of programs and data in *multi-level metaprogramming.*

In this paper we show that such a representation has the great advantage of simplifying generalization of multiply encoded data. Previous work on generalization focuses only on single-level and has not addressed the unique features of multi-level languages. A *metasystem hierarchy* is any situation where a program p_0 is manipulating (*e.g.*, interpreting, transforming) another program p_1. Program p_1 may be manipulating another program p_2, and so on. A metasystem hierarchy can be illustrated using a metasystem transition scheme as in Figure 1 [18].

The representation of programs and data has been discussed especially in the context of logic programming [2]. The present work addresses these open issues in the context of S-graph-n — a simple functional language which provides primitives for manipulating metacode and metavariables. S-graph-n features numeral-indexed constructs where the numeral indicates the

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

†Dept. of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, email: `glueck@diku.dk`.

‡Computer Science Dept., Oklahoma State University, 219 Mathematical Sciences, Stillwater, OK 74078, USA, email: `hatcliff@a.cs.okstate.edu`.

§Dept. of Mathematics and Physics, Royal Veterinary and Agricultural University, Thorvaldsensvej 50, DK-1871 Frederiksberg C, Denmark, email: `jesper@dina.kvl.dk`.

level	$0:$	$p_0:$ ————————————
level	$1:$	$p_1:$ ———————————
$\vdots$		$\vdots$
level	$n-1:$	$p_{n-1}:$ —————————
level	$n:$	$p_n:$ ————————

Figure 1: MST Scheme of a metasystem hierarchy

Programs:

prog	::=	*def**
def	::=	(`DEFINE` (*fname* $name_1 \ldots name_n$) *tree*)

Trees:

tree	::=	(`IF` *cntr tree tree*) | (`LET` *name exp tree*) | (`CALL` (*fname arg**)) | *exp*
cntr	::=	(`CONS?` *arg name name*) | (`EQA?` *arg arg*)
exp	::=	($\texttt{CONS}^n$ *exp exp*) | *arg*
arg	::=	($\texttt{ATOM}^n$ *atom*) | (`PV` *name*) | ($\texttt{MV}^n$ *h name*) *for any* $n, h \geq 0$

Figure 2: Syntax of S-Graph-n (excerpts)

the number of times that the program construct has been metacoded. The encoding of programs as data is called metacoding [18]. For example,

$$(\texttt{CONS}^2\ (\texttt{ATOM}^2\ \texttt{a})\ (\texttt{ATOM}^2\ \texttt{b})) \tag{1}$$

is an expression encoded twice. The area of online program specialization (partial deduction, supercompilation) has seen a recent flurry of activity on generalization and termination properties [1, 4, 13, 14, 18, 19]. Two popular methods used in *single-level systems* are: most specific generalization and the homeomorphic embedding relation. Our goal is to clarify issues of generalization and termination across multi-level hierarchies of programs.

2 S-Graph-n

In this section we present the language S-Graph-n and its meta-programming concepts. The full language is given in [10]; here we only consider a language fragment.

2.1 Syntax

Figure 2 presents excerpts of S-Graph-n — a first-order, functional programming language restricted to tail-recursion. A program is a list of function definitions where each function body is built from a few elements: conditionals `IF`, local bindings `LET`, function calls `CALL`, program variables (`PV` *name*), and data constructed from the indexed components $\texttt{CONS}^n$, $\texttt{ATOM}^n$, and $\texttt{MV}^n$. The components $\texttt{CONS}^n$ and $\texttt{ATOM}^n$ are the usual constructors for S-expressions, $\texttt{MV}^n$ is a constructor for metavariables (their use will be explained later). The full language [10] includes indexed components for each construct of the language (*e.g.*, $\texttt{IF}^n$, $\texttt{LET}^n$, *etc.*).

The test *cntr* in a conditional may update the environment. We excerpt two illustrative contractions [17] from the full language:

- (`EQA?` arg_1 arg_2) — tests the equality of two atoms arg_1, arg_2 of degree 0. If either argument is non-atomic or has a degree other than 0, then the test is undefined.
- (`CONS?` *exp h t*) — if the value of *exp* is a pair ($\texttt{CONS}^0$ val_1 val_2), then the test succeeds and variable h is bound to val_1 and variable t to val_2; fails, otherwise.

When programming in S-Graph-n, one usually takes 0 as the reference level. In the programming examples, components where indices are omitted are at level 0. Figure 3 shows a tail-recursive version of list reverse at level 0. As shorthand, program variables ($\texttt{PV}^0$ *name*) are sometimes written as *name*.

```
(define (rev x a)
  (if (cons? (pv x) (pv hd) (pv tl))
      (call (rev (pv tl) (cons (pv hd) (pv a))))
      (pv a)))
```

Figure 3: Tail-recursive list reverse

The *index* of an S-Graph-n term is the level number attached to the outermost construct.

$$(\texttt{CONS}^2\ (\texttt{ATOM}^1\ \texttt{a})\ (\texttt{ATOM}^3\ \texttt{b})) \tag{2}$$

$$(\texttt{CONS}^2\ (\texttt{ATOM}^2\ \texttt{a})\ (\texttt{ATOM}^3\ \texttt{a})) \tag{3}$$

$$(\texttt{CONS}^3\ (\texttt{ATOM}^2\ \texttt{1})\ (\texttt{ATOM}^1\ \texttt{2})) \tag{4}$$

In the components above, the index of (2) and (3) is 2, and the index of (4) is 3.

The *degree* of an S-Graph-n term is the smallest index occurring in the term. In the components above, (2) and (4) have degree 1; (3) has degree 2. Intuitively, if a component has degree n, then it has been metacoded at most n times (though some parts of the component may have been metacoded more times). As motivated in Section 1, the degree indicates to which level of a hierarchy a component belongs.

Evaluation of a S-Graph-n program yields a *value* — a closed (*i.e.*, ground) term in the set described by the following grammar. As syntactic sugar, metavariables ($\texttt{MV}^n$ h *name*) are written as $name^n_h$.

$$\begin{array}{lcl} val & \in & Values \\ val & ::= & X^n_h \mid (\texttt{ATOM}^n\ atom) \mid (\texttt{CONS}^n\ val\ val) \qquad \textit{for any } n, h \geq 0 \end{array}$$

2.2 Meta-programming concepts

Metaprogramming requires facilities for encoding programs as data objects. In a metasystem hierarchy, such as displayed in Figure 1, programs may be metacoded many times. For example, program p_n of Figure 1 must be metacoded (directly or indirectly) n times, since n systems lie above it in the hierarchy. Metasystem hierarchies are a cornerstone of Turchin's approach [18]: the construction of hierarchies of arbitrary height is taken as the basis for program analysis and transformation (in contrast to logics and mathematics which usually deal with two-level hierarchies).

To illustrate the *space consumption problem* in metasystem hierarchies, consider a straightforward strategy for encoding programs where μ is the metacoding function.

$$\begin{array}{rcl} \mu\{(\texttt{CONS}\ exp_1\ exp_2)\} & = & (\texttt{CONS}\ (\texttt{ATOM}\ \texttt{CONS})\ (\texttt{CONS}\ \mu\{exp_1\}\ \mu\{exp_2\})) \\ \mu\{(\texttt{ATOM}\ atom)\} & = & (\texttt{CONS}\ (\texttt{ATOM}\ \texttt{ATOM})\ (\texttt{ATOM}\ atom)) \end{array}$$

Using this strategy, the expression `(CONS (ATOM a) (ATOM b))` metacoded twice is

$$\begin{array}{rcl} \mu\{\mu\{(\texttt{CONS (ATOM a) (ATOM b)})\}\} & = & \texttt{(CONS (ATOM CONS) (CONS (CONS (ATOM ATOM) (ATOM CONS)) (CONS (ATOM} \\ & & \texttt{CONS) (CONS (CONS (ATOM CONS) (CONS (CONS (ATOM ATOM) (ATOM ATOM))} \\ & & \texttt{(CONS (ATOM ATOM) (ATOM a)))) (CONS (ATOM CONS) (CONS (CONS (ATOM ATOM)} \\ & & \texttt{(ATOM ATOM)) (CONS (ATOM ATOM) (ATOM b))))))))} \end{array}$$

The naive encoding is straightforward, but leads to an exponential growth of expressions, and yields terms that are harder to reason about. In addition, various specialization procedures (*e.g.*, generalization algorithms) that repeatedly traverse metacode will suffer a significant degrading effect using the metacoding above. There is even the risk that naive generalization of structures containing multiply encoded data such as those above will destroy some encoding structures. Level indexed data is conceptually simpler and easier to reason about [10].

Metacoding The *indexed constructs* of S-Graph-n provide a concise and natural representation of programs as data. A program component is metacoded by simply increasing its numerical index. This is formalized by the metacoding function μ given below (we show only the cases for data constructors).

$$\begin{array}{rcl} \mu\{X_h^n\} & = & X_h^{n+1} \\ \mu\{(\mathtt{ATOM}^n\ atom)\} & = & (\mathtt{ATOM}^{n+1}\ atom) \\ \mu\{(\mathtt{CONS}^n\ sgn_1\ sgn_2)\} & = & (\mathtt{CONS}^{n+1}\ \mu\{sgn_1\}\ \mu\{sgn_2\}) \end{array}$$

The injectivity of μ ensures that metacoded programs can be uniquely *demetacoded. Repeated metacoding* given by iterated application of μ (denoted μ^i for some $i \geq 0$) avoids the exponential space explosion characteristic of naive metacoding strategies. Repeated metacodings form a hierarchy of values

$$\mu^0\{\mathit{Values}\} \supseteq \mu^1\{\mathit{Values}\} \supseteq \mu^2\{\mathit{Values}\} \ldots$$

where for any S, $\mu\{S\}$ denotes the set obtained by element-wise application of μ. This hierarchy is described by the following grammar (*i.e.*, $\mu^n\{\mathit{Values}\} = \mathit{Values}[n]$).

$$\begin{array}{lcl} val^n & \in & \mathit{Values}[n] \\ val^n & ::= & X_h^{n+r} \mid (\mathtt{ATOM}^{n+r}\ atom) \mid (\mathtt{CONS}^{n+r}\ val^n\ val^n) \qquad \textit{for any } r, h \geq 0 \end{array}$$

In situations where one has a hierarchy of metasystems as in Figure 1, $\mathit{Values}[n]$ describes the set of values being manipulated by the program p_n.

Configuration values The description of *sets of values* can be further refined by the use of *metavariables.* A metavariable X_h^n with *elevation index* h ranges over the set $\mathit{Values}[n+h+1]$. $\mathit{Values}[n+h+1]$ is also called the *domain* of X_h^n. [1]

$$domain(X_h^n) = \mathit{Values}[n+h+1]$$

The set of S-Graph-n values is extended to form *configuration values.* Note that except for metavariables, all values are encoded at least once. This reflects the situation that a metaprogram p_n manipulates pieces of metacode (*e.g.*, program fragments) that are encoded at least $n+1$ times (*i.e.*, the object level is at least one level lower).

$$\begin{array}{lcl} cval^n & \in & \mathit{Cvalues}[n] \\ cval^n & ::= & X_h^{n+r} \mid (\mathtt{ATOM}^{n+m}\ atom) \mid (\mathtt{CONS}^{n+m}\ cval^n\ cval^n) \qquad \textit{for any } r, h \geq 0,\ m \geq 1 \end{array}$$

Intuitively, one can think of configuration values $cval^n$ as a description language for characterizing metacoded data patterns occurring in a metasystem hierarchy at level n. Formally, each configuration value $cval^n \in \mathit{Cvalues}[n]$ denotes a set $[\![cval^n]\!]^n \subseteq \mathit{Values}[n+1]$. Note that only X_h^n are considered as metavariables on level n; metavariables with degree $n+m, m \geq 1$ are treated as encoded constructors.

$$\begin{array}{rcl} [\![\cdot]\!]^n & : & \mathit{Cvalues}[n] \to \wp(\mathit{Values}[n+1]) \\ [\![X_h^n]\!]^n & = & domain(X_h^n) \\ [\![X_h^{n+m}]\!]^n & = & \{X_h^{n+m}\} \\ [\![(\mathtt{ATOM}^{n+m}\ atom)]\!]^n & = & \{(\mathtt{ATOM}^{n+m}\ atom)\} \\ [\![(\mathtt{CONS}^{n+m}\ val_1\ val_2)]\!]^n & = & \{(\mathtt{CONS}^{n+m}\ cval_1\ cval_2) \mid val_1 \in [\![cval_1]\!], val_2 \in [\![cval_2]\!]\} \end{array}$$

Consider the following example showing the set represented by $(\mathtt{CONS}^2\ (\mathtt{ATOM}^3\ \mathtt{a})\ X_1^0)$.

$$\begin{array}{rcl} [\![(\mathtt{CONS}^2\ (\mathtt{ATOM}^3\ \mathtt{a})\ X_1^0)]\!]^0 & = & \{(\mathtt{CONS}^2\ (\mathtt{ATOM}^3\ \mathtt{a})\ (\mathtt{ATOM}^2\ \mathtt{a})), (\mathtt{CONS}^2\ (\mathtt{ATOM}^3\ \mathtt{a})\ (\mathtt{ATOM}^3\ \mathtt{a})), \ldots, \\ & & (\mathtt{CONS}^2\ (\mathtt{ATOM}^3\ \mathtt{a})\ (\mathtt{ATOM}^2\ \mathtt{b})), (\mathtt{CONS}^2\ (\mathtt{ATOM}^3\ \mathtt{a})\ (\mathtt{ATOM}^3\ \mathtt{b})), \ldots, \} \end{array}$$

[1] The reader should be warned that our definition of metavariable degree differs slightly from other work [8, 20]. In those works, our metavariable of degree n has degree $n+1$.

Most specific domain index Clearly, for every $cval^n$ there exists an i such that $[\![cval^n]\!]^n \subseteq domain(X_i^n)$ (one can always take $i = 0$). However, since domains $Values[\cdot]$ form a hierarchy, there exists a maximal k such that $[\![cval^n]\!]^n \subseteq domain(X_k^n)$ and $[\![cval^n]\!]^n \not\subseteq domain(X_{k+1}^n)$. Intuitively, $domain(X_k^n)$ describes the most specific value domain containing $[\![cval^n]\!]^n$. Therefore, we refer to k as the *most specific domain index of* $cval^n$ *at level* n (denoted $msdi^n(cval^n)$).

$$
\begin{array}{rcl}
msdi^n(\cdot) & : & Cvalues[n] \rightarrow Number \\
msdi^n(X_h^n) & = & h \\
msdi^n(X_h^{n+m}) & = & m-1 \\
msdi^n((\texttt{ATOM}^{n+m}\ atom)) & = & m-1 \\
msdi^n((\texttt{CONS}^{n+m}\ cval_1\ cval_2)) & = & \min(m-1, msdi^n(cval_1), msdi^n(cval_2))
\end{array}
$$

Consider the following examples of most specific domain indices.

$$msdi^0((\texttt{CONS}^2\ (\texttt{ATOM}^3\ \texttt{a})\ X_1^0)) = 1 \tag{5}$$

$$msdi^0((\texttt{CONS}^5\ (\texttt{ATOM}^3\ \texttt{a})\ X_1^3)) = 2 \tag{6}$$

$$msdi^3((\texttt{CONS}^5\ (\texttt{ATOM}^6\ \texttt{a})\ (\texttt{ATOM}^4\ \texttt{b}))) = 0 \tag{7}$$

3 Generalization of Configuration Values

The explicit identification of the hierarchical position of values given by configuration values is a *key* component of the strategy for successful self-application of online specializers given in *e.g.*, [20, 10]. This work describes how configuration values and metavariables are used to propagate information during specialization.

An equally important component is the *termination strategy*: avoiding an infinite sequence of specialization states by folding back to some previously encountered specialization state. This may require identifying the information common to two specialization states. Since two specialization states must now be described by one, precision may be lost. Thus, one desires the generalization to be as specific as possible.

In our setting, specialization states are described by configuration values that represent sets of values. Information common to $cval_1^n$ and $cval_2^n$ is given by $[\![cval_1^n]\!]^n \cup [\![cval_2^n]\!]^n$. Using configuration values as description language, the goal of generalization is find a description $cval_g^n$ such that $[\![cval_1^n]\!]^n \cup [\![cval_2^n]\!]^n \subseteq [\![cval_g^n]\!]^n$ and $[\![cval_g^n]\!]^n$ is as small (precise) as possible. In other words, the least loss of information occurs.

These concepts can be formalized by extending familiar concepts of *most-specific generalization*, as applied previously to program specialization in *e.g.*, [15], to level-index constructors and elevated metavariables.

A *binding* $X_h^n := cval$ is a metavariable/configuration value pair. A binding is *elevation-compatible* when $h \leq msdi^n(cval)$. Note that this ensures $[\![cval]\!]^n \subseteq domain(X_h^n)$. A *substitution at level* n, $\theta^n = \{X_{1\,h_1}^{\ n} := cval_1^n, \ldots, X_{t\,h_t}^{\ n} := cval_t^n\}$, is a finite set of bindings such that the metavariables $X_{i\,h_i}^{\ n}$ are pairwise distinct. A substitution is *elevation-compatible* when all of its bindings are elevation-compatible. An *instance* of $cval^n$ is a value of the form $cval^n\theta^n$ where θ^n is a substitution at level n.

Definition 1 (renaming,generalization,msg)

1. *A* renaming substitution *for* $cval^n$ *is a substitution* $\{X_{1\,h_1}^{\ n} := Y_{1\,h_1}^{\ n}, \ldots, X_{t\,h_t}^{\ n} := Y_{t\,h_t}^{\ n}\}$ *such that there is a binding for every metavariable occurring in* $cval^n$ *and* $Y_{i\,h_i}^{\ n}$ *are pairwise distinct.*
2. *A* generalization *of* $cval_1^n, cval_2^n$ *at level* n *is a triple* $(cval_{gen}^n, \theta_1^n, \theta_2^n)$ *where* $cval_1^n \equiv cval_{gen}^n\theta_1^n$ *and* $cval_2^n \equiv cval_{gen}^n\theta_2^n$.
3. *A generalization* $(cval_{gen}^n, \theta_1^n, \theta_2^n)$ *of* $cval_1^n$ *and* $cval_2^n$ *at level* n *is* most specific *(msg) if for every generalization* $(cval_{gen}'^n, \theta_1'^n, \theta_2'^n)$ *of* $cval_1^n$ *and* $cval_2^n$, $cval_{gen}^n$ *is an instance of* $cval_{gen}'^n$.

$$\begin{pmatrix} \tau \\ \{X_h^n := X_i^{n+m}\} \cup \theta \\ \{X_h^n := X_i^{n+m}\} \cup \theta' \end{pmatrix} \rightarrow \begin{pmatrix} \tau\{X_h^n := X_i^{n+m}\} \\ \theta \\ \theta' \end{pmatrix}$$

$$\begin{pmatrix} \tau \\ \{X_h^n := (\mathtt{ATOM}^{n+m}\ atom)\} \cup \theta \\ \{X_h^n := (\mathtt{ATOM}^{n+m}\ atom)\} \cup \theta' \end{pmatrix} \rightarrow \begin{pmatrix} \tau\{X_h^n := (\mathtt{ATOM}^{n+m}\ atom)\} \\ \theta \\ \theta' \end{pmatrix}$$

$$\begin{pmatrix} \tau \\ \{X_h^n := (\mathtt{CONS}^{n+m}\ v\ w)\} \cup \theta \\ \{X_h^n := (\mathtt{CONS}^{n+m}\ v'\ w')\} \cup \theta' \end{pmatrix} \rightarrow \begin{pmatrix} \tau\{X_h^n := (\mathtt{CONS}^{n+m}\ Y_i^n\ Z_j^n)\} \\ \{Y_i^n := v, Z_j^n := w\} \cup \theta \\ \{Y_i^n := v', Z_j^n := w'\} \cup \theta' \end{pmatrix}$$

$$\textit{where } Y, Z \textit{ are fresh and } \begin{array}{rcl} i & = & \min(msdi^n(v), msdi^n(v')) \\ j & = & \min(msdi^n(w), msdi^n(w')) \end{array}$$

$$\begin{pmatrix} \tau \\ \{X_h^n := v, Y_h^n := v\} \cup \theta \\ \{X_h^n := v', Y_h^n := v'\} \cup \theta' \end{pmatrix} \rightarrow \begin{pmatrix} \tau\{X_h^n := Y_h^n\} \\ \{Y_h^n := v\} \cup \theta \\ \{Y_h^n := v'\} \cup \theta' \end{pmatrix}$$

Figure 4: Computation of most specific generalization on level n

Definition 2 (Algorithm of msg) *For any $cval_1^n, cval_2^n \in Cvalues[n]$, the most specific generalization $msg^n(cval_1^n, cval_2^n)$ is computed by exhaustively applying the rewrite rules of Figure 4 to the initial triple*

$$(X_h^n, \{X_h^n := cval_1^n\}, \{X_h^n := cval_2^n\}) \qquad h = \min(msdi^n(cval_1^n), msdi^n(cval_2^n))$$

where X_h^n is a variable not occurring in $cval_1^n$ and $cval_2^n$.

Theorem 1 *For any $cval_1^n, cval_2^n \in Cvalues[n]$, the procedure (Def. 2) is indeed an algorithm, i.e. it terminates and the result $msg^n(cval_1^n, cval_2^n)$ is the most specific generalization at level n.*

Proof: (sketch)

1. *Termination of the algorithm.* We prove that in every rewrite rule the size of the substitutions Θ, Θ' or the size of the values in their bindings decreases. Since values are well-founded, the rewrite rules can only be applied finitely many times.
2. *Soundness of algorithm.* We prove that the algorithm always computes a generalization: the initial triple is a generalization and after each rewrite rule the new triple is still a generalization.
3. *Computation of msg.* We prove by contradiction that the generalization computed by the algorithm is an msg. Assume that the generalization $cval_{gen}^n$ computed by the algorithm is not an msg, then there exists an msg $cval_{msg}^n$ such that $cval_{msg}^n = cval_{gen}^n \theta^n$ where θ^n is not a renaming substitution. We show that θ^n must be a renaming substitution if $cval_{gen}^n$ is computed by the algorithm (there will always be a rewrite rule that applies and makes the substitution a renaming).

The following are two examples of the most specific generalization at level 2: ∎

$$\begin{array}{rl} msg^2((\mathtt{CONS}^3\ X_2^2\ (\mathtt{ATOM}^4\ \mathtt{a})), (\mathtt{CONS}^3\ Y_1^2\ (\mathtt{ATOM}^3\ \mathtt{a}))) = & ((\mathtt{CONS}^3\ Z_1^2\ R_0^2), \\ & \{Z_1^2 := X_2^2, R_0^2 := (\mathtt{ATOM}^4\ \mathtt{a})\}, \\ & \{Z_1^2 := Y_1^2, R_0^2 := (\mathtt{ATOM}^3\ \mathtt{a})\}) \end{array}$$

$$
\begin{array}{ll}
X_h^n \trianglelefteq^n Y_h^n & [VAR] \\
X_h^{n+m} \trianglelefteq^n X_h^{n+m} & [VAR\text{-}MC] \\
(\mathtt{ATOM}^n\ atom) \trianglelefteq^n (\mathtt{ATOM}^n\ atom) & [ATOM] \\
\dfrac{cval_1 \trianglelefteq^n cval_1' \qquad cval_2 \trianglelefteq^n cval_2'}{(\mathtt{CONS}^{n+m}\ cval_1\ cval_2) \trianglelefteq^n (\mathtt{CONS}^{n+m}\ cval_1'\ cval_2')} & [CONS] \\
\dfrac{cval \trianglelefteq^n cval_1}{cval \trianglelefteq^n (\mathtt{CONS}^{n+m}\ cval_1\ cval_2)} & [DIVE\text{-}L] \\
\dfrac{cval \trianglelefteq^n cval_2}{cval \trianglelefteq^n (\mathtt{CONS}^{n+m}\ cval_1\ cval_2)} & [DIVE\text{-}R]
\end{array}
$$

Figure 5: Homeomorphic embedding on level n

$$msg^2((\mathtt{CONS}^3\ Y_2^2\ Y_2^2), (\mathtt{CONS}^3\ Z_2^3\ Z_2^3)) = ((\mathtt{CONS}^3\ X_0^2\ X_0^2), \{X_0^2 := Y_2^2\}, \{X_0^2 := Z_2^3\})$$

4 Embedding of Configuration Values

The homeomorphic embedding relation known from term algebra [6] can be used to ensure that unfolding during a specialization process does not proceed infinitely. If $cval_1^n \trianglelefteq^n cval_2^n$ then this suggests that $cval_2^n$ might arise from $cval_1^n$ and may lead to some infinitely continuing process and that the transformation should be stopped. Variants of this relation are used in termination proofs for term rewriting systems and for ensuring termination of partial deduction and supercompilation, *e.g.* in [13, 14, 18].

In our setting, the embedding relation works on specialization states described by configuration values. Since there is only a finite number of elevations in a given problem, we strengthen the embedding relation for variables on level n (rule *VAR*) with the condition that the elevation indices are identical. In rule *VAR-MC* variables on level $n + m$ are treated as data (they are encoded variables). The rules *ATOM* and *CONS* require that the degree of the constructors is identical. Rule *DIVE-L* and *DIVE-R* detect a configuration value embedded as subterm in another larger configuration value; the other rules match components by components. It is not difficult to give an algorithm deciding whether $cval_1^n \trianglelefteq^n cval_2^n$.

Theorem 2 *The relation* $\trianglelefteq^n$ *is a well-quasi order on Cvalues[n].*
Proof: (omitted) ∎

5 Self-Applicable Specialization

The previous sections were concerned with the development of methods for dealing with multiply encoded data; this one is concerned with a practical study in a self-applicable specializer. This specializer performs constant propagation (no propagation of information as in supercompilation). First we give an example that introduces the notation, then we illustrate a case of self-application. The specializer has two arguments: `tree` and `def` where `tree` is the initial expression (metacoded once) and `def` contains the definitions of the program to be specialized. Figure 6 shows how the specialization is setup for the reverse program (Figure 3) when the first argument is known (the list `[1, 2]`) and the accumulator (`a`) is unknown (a metavariable on level 0 with elevation 0).

A multi-level example We now turn to a multi-level example. The program we consider is again the reverse program, but this time we apply a second version of the specializer (self-application) to the specialization of the program, and the first list (`x`) is replaced by a meta-variable at level 0 with elevation 1 and the accumulator (`a`) by a meta-variable at level 1 with

Initial call to the (self-applicable) specializer:

```
(let tree (call-1 (rev (cons-1 (atom-1 1) (cons-1 (atom-1 2) (atom-1 nil))) (mv 0 a)))
  (let defs (cons (cons (atom rev) (atom nil)) (cons <rev metacoded once> (atom nil)))
    (call (spec-start tree defs)))))
```

Result of specialization:

```
(cons-1 (atom-1 2) (cons-1 (atom-1 1) (pv-1 a)))
```

Figure 6: Specialization example

elevation 0. This means that `x` is a variable of the outer specializer and `a` is a variable of the inner specializer. This can be illustrated by the following MST scheme:

$$\begin{array}{lll} \text{level } 0: & \texttt{spec}_0\text{———} \; \texttt{x}_1^0 \; \text{——} & \\ \text{level } 1: & \qquad \texttt{spec}_1\text{——} \; | \; \text{—} \; \texttt{a}_0^1 & \\ \text{level } 2: & \qquad\qquad \texttt{rev}(\bullet \;,\; \bullet) & \end{array}$$

We now examine some essential steps in the development of the computation. The initial call of metasystem hierarchy is shown in Figure 7. After some computation steps this leads to the configuration `(1)` which the outer specializer will memorize such that it can check whether later configurations embed this. We assume that a specializer does this for all calls, since unfolding of function calls is the only situation that can cause infinite specialization. Since our language is tail-recursive all memorized configuration will consists of a call to a function from the specializer and we will simply write `spec-*` to denote some function from the specializer. After memorizing the configuration the inner specializer (interpreted by the outer) unfolds the call to `rev`. This leads to a configuration:

```
(call (spec-*
  (call-1 (spec-* (if-2 (cons?-2 (mv 1 x) (pv-2 hd) (pv-2 tl))
                        (call-2 (rev (pv-2 tl) (cons-2 (pv-2 hd) (mv-1 0 a))))
                        (mv-1 0 a)) ...)) ...))
```

in which the inner specializer cannot continue, because to decide the outcome of the test it has to know the value of `(mv 1 x)` and this metavariable belongs to the outer specializer. This means that the outer specializer takes over and drives the configuration by splitting it into two configurations: one for each branch of the conditional. The second of these configurations is `(2)` shown in Figure 7. We have reached a point where configuration `(2)` embeds `(1)`.

The fragments represented by `...` in configuration `(1)` and `(2)` are identical and, thus, trivially embed each other. Similarly, the functions `spec-*` are the same in both configurations. So we only have to consider whether the following holds:

$$\texttt{(rev (mv 1 x) (mv-1 0 a))} \trianglelefteq^0 \texttt{(rev (mv 1 tl) (cons-2 (mv 1 hd) (mv-1 0 a)))}$$

Since both trees have the same outer constructor we use the *CONS*-rule from Figure 5, which means that we consider:

$$\texttt{(mv 1 x)} \trianglelefteq^0 \texttt{(mv 1 tl)} \;\; \textbf{and} \;\; \texttt{(mv-1 0 a)} \trianglelefteq^0 \texttt{(cons-2 (mv 1 hd) (mv-1 0 a))}$$

instead. The first of these two clearly holds because of the *VAR*-rule in Figure 5. The second holds because the *DIVE-R* rule and the *VAR*-rule (`(mv-1 0 a)` $\trianglelefteq$ `(mv-1 0 a)`). Configuration `(2)` embeds `(1)`.

Usually, when a late configuration embeds an early configuration during specialization this means that there is a risk that specialization will not terminate. So therefore we assume that our

Initial call to the self-applicable specializer:

```
(let tree
  (let-1 tree (call-2 (rev (mv 1 x) (pv-1 0 a)))
    (let-1 defs (cons-1 (cons-1 (atom-1 rev) (atom-1 nil))
                        (cons-1 <rev metacoded twice> (atom-1 nil)))
        (call-1 (spec-start (pv-1 tree) (pv-1 defs)))))
  (let defs <spec metacoded once>
    (call (spec-start tree env defs))))
```

Initial configuration after evaluation (unfolding) of the let-expressions:

```
(call (spec-* (call-1 (spec-* (call-2 (rev (mv 1 x) (mv-1 0 a)))) ...) ...))   (1)
```

Configuration after some steps of computation (embedding `(1)`*):*

```
(call
  (spec-*
    (call-1
      (spec-* (call-2 (rev (mv 1 tl) (cons-2 (mv 1 hd) (mv-1 0 a)))) ...))     (2)
    ...))
```

Generalized configuration (msg of `(1)` *and* `(2)`*):*

```
(call (spec-* (call-1 (spec-* (call-2 (rev (mv 1 x) (mv 1 z)))) ...) ...))
```

Figure 7: Specialization example showing embedding and generalization

specializer in the example replaces the early configuration by the most specific generalization of the two configurations. For our example Figure 7 shows the msg of `(1)` and `(2)` that is obtained by the the method shown in Figure 4.

6 Related Work

The ideas present in this paper have been heavily influenced by three concepts present in Turchin's work [17]: metacoding, metavariables, and metasystem transition. Subsequently, these concepts have been formalized [8] and studied in different contexts, *e.g.* [18].

Representing and reasoning about object level theories is an important field in logic and artificial intelligence (e.g. different encodings have been discussed in [11]) and has led to the development of logic languages that support declarative metaprogramming (*e.g.* the programming language Gödel [12]). In recent work [10], we proposed a simple language S-graph-n to study meta-programming aspects of self-applicable online program specialization.

MetaML, a statically typed multi-level language for hand-writing multi-level generating extensions was introduced in [16]. Another multi-level programming language is *Alloy* [3], a logic language which provides facilities for deductions at different meta-levels and a proof system for interleaving computations at different metalevels. A program generator for multi-level specialization [9] uses a functional language extended with multiple-binding times as representation of multi-level generating extensions. They allow the design and implementation of generative software [7].

Most specific generalization and the homeomorphic embedding relation are known from term algebra [6]. Variants of this relation are used in termination proofs for term rewrite systems [6] and for ensuring local termination of partial deduction [4]. After it was taken up in [14], it has inspired more recent work [1, 13, 19, 18].

7 Conclusion

Our goal was to clarify foundational issues of generalization and termination in hierarchies of programs with multiple encoded data. We examined two popular methods, most specific generalization and the homeomorphic embedding relation, proved their properties and illustrated their working with simple examples. It is clear that several challenging problems lie ahead: the implementation of a full system based on the approach advocated in this paper and formalization of other termination and generalization strategies for multi-level hierarchies of programs. We believe that the work presented in this paper is a solid basis for further work on multi-level program hierarchies and their efficient implementation on the computer.

References

[1] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven partial evaluation of functional logic programs. In H.R. Nielson, editor, *European Symposium on Programming*, Lecture Notes in Computer Science, pages 46–61. Springer-Verlag, 1996.

[2] K. Apt and F. Turini *Meta-Logics and Logic Programming*, MIT Press, 1995.

[3] J. Barklund. A basis for a multilevel metalogic programming language. Technical Report 81, Uppsala University, Dept. of Computing Science, 1994.

[4] R. Bol. Loop checking in partial deduction. *J of Logic Programming*, 16(1&2):25–46, 1993.

[5] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[6] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 244–320. Elsevier, 1992.

[7] U. Eisenecker. Generative programming with C++. In H. Mössenböck, editor, *Modular Programming Languages*, volume 1204 of *Lecture Notes in Computer Science*, pages 351–365. Springer-Verlag, 1997.

[8] R. Glück. On the mechanics of metasystem hierarchies in program transformation. In M. Proietti, editor, *Logic Program Synthesis and Transformation (LoPSTr'95)*, volume 1048 of *Lecture Notes in Computer Science*, pages 234–251. Springer-Verlag, 1996.

[9] R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.

[10] J. Hatcliff and R. Glück. Reasoning about hierarchies of online specialization systems. In Danvy et al. [5].

[11] P. Hill and J. Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, 1994.

[12] P. Hill and J.W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[13] M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In Danvy et al. [5], pages 263–283.

[14] M.H. Sørensen, R. Glück, and N.D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[15] M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.

[16] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, 1997.

[17] V.F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[18] V.F. Turchin. Metacomputation: Metacomputation: metasystem transitions plus supercompilation. In Danvy et al. [5].

[19] V.F. Turchin. On generalization of lists and strings in supercompilation. Technical Report CSc. TR 96-002, City College of the City University of New York, 1996.

[20] V.F. Turchin and A.P. Nemytykh. Metavariables: their implementation and use in program transformation. CSc. TR 95-012, City College of the City University of New York, 1995.