# Generalization in Hierarchies of Online Program Specialization Systems

Robert Glück[1], John Hatcliff[2], and Jesper Jørgensen[3]

[1] University of Copenhagen, Dept. of Computer Science, Universitetsparken 1, DK-2100 Copenhagen, Denmark. Email: glueck@diku.dk
[2] Kansas State University, Dept. of Computing and Information Sciences, 234 Nichols Hall, Manhattan, KS 66506, USA. Email: hatcliff@cis.ksu.edu
[3] Royal Veterinary and Agricultural University, Dept. of Mathematics and Physics, Thorvaldsensvej 50, DK-1871 Frederiksberg, Denmark. Email: jesper@dina.kvl.dk

**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. After developing and formalizing the basic metaprogramming concepts, we extend two basic methods to multiply encoded data: most specific generalization and the homeomorphic embedding relation. Examples and experiments with the initial design of an online specializer illustrate their use in hierarchies of online program specializers.

## 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.* [4, 11, 24]. Representing and reasoning about object level theories is an important field in logic and artificial intelligence and has led to the development of logic languages that facilitate metaprogramming (*e.g.* the logic languages Gödel [15], Reflective Prolog [6], $\lambda$-Prolog [20]).

Our goal is to lay the foundations for a *multi-level metaprogramming environment* that fully addresses the unique requirements of multi-level program hierarchies and to develop techniques for efficient computation on all levels of a metasystem hierarchy. 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.

Writing meta-interpreters is a well-known technique to enhance the expressive power of logic programs [23, 3]. The approach of applying a program transformer to another program transformer—not just to an ordinary program—has

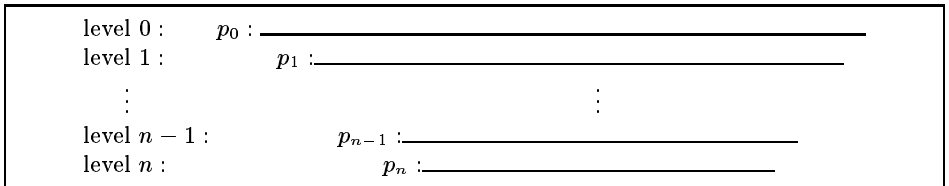



**Fig. 1.** MST Scheme of a metasystem hierarchy

been put to good use in the area of program specialization [16, 7] (a notable example are the Futamura projections). Common to these approaches is the construction and manipulation of two or more program levels.

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 [26]. 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. The representation of programs and data has been studied especially in the context of logic programming [3] (mostly two-level metasystem hierarchies).

Among others, exponential space consumption, time inefficiencies, and lack of representation invariance of meta-level operations are problems to be tackled. We now shortly illustrate some of these problems.

*Example 1.* To illustrate the **space consumption problem** in metasystem hierarchies, consider a straightforward strategy for encoding S-expressions (as known from Lisp) where $\mu$ is the metacoding function.

$$\mu\{(\texttt{CONS}\ exp_1\ exp_2)\} = (\texttt{CONS}\ (\texttt{ATOM}\ \texttt{CONS})\ (\texttt{CONS}\ \mu\{exp_1\}\ \mu\{exp_2\}))$$
$$\mu\{(\texttt{ATOM}\ atom)\} = (\texttt{CONS}\ (\texttt{ATOM}\ \texttt{ATOM})\ (\texttt{ATOM}\ atom))$$

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

$\mu\{\mu\{$`(CONS (ATOM a) (ATOM b))`$\}\} =$ `(CONS (ATOM CONS) (CONS (CONS (ATOM ATOM) (ATOM CONS)) (CONS (ATOM CONS) (CONS (CONS (ATOM CONS) (CONS (CONS (ATOM ATOM) (ATOM ATOM)) (CONS (ATOM ATOM) (ATOM a)))) (CONS (ATOM CONS) (CONS (CONS (ATOM ATOM) (ATOM ATOM)) (CONS (ATOM ATOM) (ATOM b))))))))`

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 from a significant computational degrading effect as can be seen from this example.

The area of online program transformation (e.g., partial deduction, supercompilation) has seen a recent flurry of activity, e.g. [2, 5, 19, 21, 22, 26, 27]. Recently well-quasi orders in general, and homeomorphic embedding in particular,

have gained popularity to ensure termination of program analysis, specialization and other program transformation techniques because well-quasi orders are strictly more powerful [18] than a large class of approaches based on well-founded orders. Despite the progress that has occurred during the last years, metasystem hierarchies still pose an challenge to online transformation and metaprogramming.

*Example 2.* To illustrate the **invariance problem**, consider the influence encoding has on the termination behavior of online transformer relying on the homeomorphic embedding relation $\trianglelefteq$ defined for S-expressions:

$$\begin{array}{ll} (\texttt{ATOM}\ a) \trianglelefteq (\texttt{ATOM}\ b) & \Leftarrow a = b \\ (\texttt{CONS}\ s_1\ s_2) \trianglelefteq (\texttt{CONS}\ t_1\ t_2) & \Leftarrow s_i \trianglelefteq t_i \text{ for all } i \in \{1,2\} \\ s \trianglelefteq (\texttt{CONS}\ t_1\ t_2) & \Leftarrow s \trianglelefteq t_i \text{ for some } i \in \{1,2\} \end{array}$$

Consider the following two terms which may occur during direct transformation of a program (e.g., as arguments of a function or predicate). The embedding relation signals the transformer that it can continue:

$$(\texttt{ATOM ATOM}) \not\trianglelefteq (\texttt{CONS}\ (\texttt{ATOM a})\ (\texttt{ATOM a}))$$

Assume that we insert an interpreter (or another metaprogram) that works on encoded terms. Now the embedding relation signals the transformer that it should stop immediately:

$$\mu\{(\texttt{ATOM ATOM})\} \trianglelefteq \mu\{(\texttt{CONS}\ (\texttt{ATOM a})\ (\texttt{ATOM a}))\}$$

In short, encoding the terms leads to premature termination. Ideally, a termination condition is invariant under encoding, so that termination patterns are detected reliably regardless of the encoding of terms. Similar problems occur with other operations popular with online transformers. Existing transformers are only geared toward the direct transformation of programs. They often fail to perform well on an extra metasystem level, so that several refinements have been suggested, e.g. [29]. For a discussion of the invariance problem see also [17].

Higher-order terms provide a suitable data structure for representing programs in a higher-order abstract syntax. Higher-order logic programming utilizes higher-order logic as the basis for computation, e.g. [20]. The primary advantage of higher-order abstract syntax is that it simplifies substitution and reasoning about terms up to alpha-conversion. However, we believe that there are a number of theoretical difficulties that one would encounter when using high-order logic as the basis for multi-level metaprogramming. For instance, the question of whether or not a unifier exists for an arbitrary disagreement set is known to be undecidable. Moreover, extending online program transformers, e.g. partial deducers, to higher-order logic programming languages even for single-level transformation seems very challenging and has not yet been attempted. Finally, it is unclear how much higher-order abstract syntax would help the exponential explosion due to multiple encodings in self-application situations since its strength is the encoding of variable bindings – not the encoding of arbitrary syntax constructors.

*Programs:*

$prog ::= def^*$
$def ::= (\texttt{DEFINE}\ (fname\ name_1 \ldots name_n)\ tree)$

*Trees:*

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

**Fig. 2.** Syntax of S-Graph-$n$ (excerpts)

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

**Fig. 3.** Tail-recursive list reverse

In recent work [13], we proposed a simple language S-graph-$n$ to study metaprogramming aspects of self-applicable online program specialization from a new angle. 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*. Previous work on generalization focuses only on single-level and has not addressed the unique features of metasystem hierarchies. In particular, we are not aware of another line of work that has approached foundational and practical issues involved in hierarchies of online specialization systems based on a multi-level metaprogramming environment.

**Organization** After presenting the subset of S-Graph-$n$ needed for our investigation in Section 2, we develop and formalize the basic metaprogramming concepts in Section 3. Generalization of configuration values is addressed in Section 4 and a well-quasi order for ensuring termination is given in Section 5. In Section 6 we report on the initial design and first experiments with an online specializer for S-Graph-$n$ to substantiate the claims. The papers concludes with a discussion of related work in Section 7 and future work in Section 8.

## 2 S-Graph-$n$

The choice of the subject language is important for studying foundational problems associated with hierarchies of online transformation systems. On the one hand the language must be simple enough, so that it can become the object of theoretical analysis is and program transformation, and, on the other hand, it

must be rich enough to serve as a language for writing meta-programs which can substantiate theoretical claims by computer experiments.

In this section we present the language S-Graph-$n$ and its meta-programming concepts. The full language is given in [13]; here we only need to consider a language fragment. The language seems well-suited for our purposes. Among others, a subset of the language was successfully used for clarifying basic concepts of supercompilation, e.g. [1, 12].

### 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 [13] includes indexed components for each construct of the language (*e.g.*, $\texttt{IF}^n$, $\texttt{LET}^n$, *etc.*).

### 2.2 Semantics

The operational semantics of the language is given elsewhere [13]. Here we describe it only informally. Evaluation of a S-Graph-$n$ program yields a *value* — a closed (*i.e.*, ground) term in the set described by the grammar shown in Figure 4. Values are either metavariables, atoms, or `CONS`-cells indexed by $n$. As syntactic sugar, metavariables ($\texttt{MV}^n$ $h$ *name*) are written as $name^n_h$. Intuitively, constructs indexed by $n \geq 1$ represent encoded values from higher levels.

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

**Fig. 4.** S-Graph-$n$ values (excerpts)

The test *cntr* in a conditional may update the environment. We excerpt three illustrative contractions [25] from the full language. They test and deconstruct values.

- (`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`? *arg* $h$ $t$) — if the value of *arg* 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 $val_2$; otherwise it fails.

– (MV? *exp h x*) — if the value of *exp* is a metavariable ($\texttt{MV}^0$ $val_1$ $val_2$), then the test succeeds and variable $h$ is bound to elevation $val_1$ and variable $x$ to name $val_2$; otherwise it fails.

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 written as *name*.

## 3 Meta-Programming Concepts

In this section we show how the primitives of the language provide support for multiple encodings of programs and then develop the basic meta-programming concepts [28] for S-Graph-$n$. We formalize the concepts and illustrate them with several examples.

### 3.1 Metacoding

The *indexed constructs* of S-Graph-$n$ provide a concise and natural representation of programs as data. A program component is encoded by simply increasing its numerical index. This is formalized by the metacoding function $\mu$ given below (we show only the cases for data constructors). The injectivity of $\mu$ ensures that metacoded programs can be uniquely *demetacoded*.

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

**Fig. 5.** S-Graph-$n$ metacoding

Repeated metacoding given by iterated application of $\mu$ (denoted $\mu^i$ for some $i \geq 0$) avoids the exponential space explosion characteristic of naive metacoding strategies (cf. Example 1). Thereby, the degrading effect such encoding strategies have on operations traversing metacode is avoided (e.g., generalization, homeomorphic embedding).

*Example 3.* The following example illustrate the representation of encoded data in S-Graph-$n$.

$$\begin{aligned}
\mu^1\{(\texttt{CONS (ATOM a) (ATOM b)})\} &= (\texttt{CONS}^1\ (\texttt{ATOM}^1\ \texttt{a})\ (\texttt{ATOM}^1\ \texttt{b})) \\
\mu^2\{(\texttt{CONS (ATOM a) (ATOM b)})\} &= (\texttt{CONS}^2\ (\texttt{ATOM}^2\ \texttt{a})\ (\texttt{ATOM}^2\ \texttt{b})) \\
\mu^3\{(\texttt{CONS (ATOM a) (ATOM b)})\} &= (\texttt{CONS}^3\ (\texttt{ATOM}^3\ \texttt{a})\ (\texttt{ATOM}^3\ \texttt{b})) \\
\ldots &= \ldots
\end{aligned}$$

One often needs to embed data with a lower indices as components of constructs with higher indices. The *degree* of an S-Graph-$n$ term is the smallest index occurring in a term. As motivated in Section 1, the degree indicates to which level of a hierarchy a component belongs. Intuitively, if a component has degree $n$, then it has been metacoded $n$ times (though some parts of the component may have been metacoded more times).

*Example 4.* Consider the following S-Graph-$n$ values.

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

The components (1) and (3) have degree 1 and (2) has degree 2. They have been encoded at most once and twice, respectively.

### 3.2 Hierarchy of Metacoded Values

Repeated metacoding of values induces an important property: it creates a hierarchy of sets of metacoded values.

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

where for any $S$, $\mu\{S\}$ denotes the set obtained by element-wise application of $\mu$. Each set of metacoded values is described by the grammar in Figure 6 (*i.e.*, $\mu^n\{\mathit{Values}\} = \mathit{Values}[n]$).

$$\mathit{val}^n \in \mathit{Values}[n]$$
$$\mathit{val}^n ::= X_h^{n+r} \mid (\texttt{ATOM}^{n+r}\ \mathit{atom}) \mid (\texttt{CONS}^{n+r}\ \mathit{val}^n\ \mathit{val}^n) \qquad \mathit{for\ any}\ r, h \geq 0$$

**Fig. 6.** Sets of metacoded values

The following property formalizes the characteristic of $\mu$.

*Property 1 (Hierarchical embedding).*
$\forall n \geq 0\,.\ \ \mathit{Values}[n] \supset \mathit{Values}[n+1]$

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 $p_n$. This observation plays an important role when we abstract from metacoded values in a hierarchy of metasystems.

*Example 5.* The values encoded in Example 3 belong to the following value sets.

$$\mu^1\{(\texttt{CONS}\ (\texttt{ATOM}\ \texttt{a})\ (\texttt{ATOM}\ \texttt{b}))\} \in \mathit{Values}[1]$$
$$\mu^2\{(\texttt{CONS}\ (\texttt{ATOM}\ \texttt{a})\ (\texttt{ATOM}\ \texttt{b}))\} \in \mathit{Values}[2]$$
$$\mu^3\{(\texttt{CONS}\ (\texttt{ATOM}\ \texttt{a})\ (\texttt{ATOM}\ \texttt{b}))\} \in \mathit{Values}[3]$$
$$\ldots$$

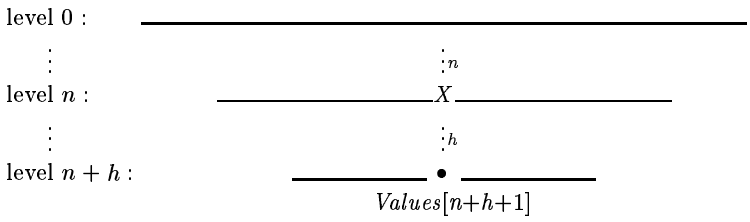



**Fig. 7.** Positioning of metavariable $X_h^n$ in an MST Scheme

### 3.3 Metavariables and their Domain

The description of *sets of values* can be further refined by the use of *metavariables*. A metavariable is an abstraction that represents a set of values. A metavariable $X_h^n$ has three attributes which determine its semantics: *degree*, *domain*, *elevation*.

$$\begin{aligned} \mathit{degree}(X_h^n) &= n \\ \mathit{domain}(X_h^n) &= \mathit{Values}[n+h+1] \\ \mathit{elevation}(X_h^n) &= h \end{aligned}$$

*Degree* $n$, as we have seen before, indicates the number of times that a metavariable has been metacoded. *Domain* is the set of values over which a metavariable ranges. *Elevation* $h$ restricts the domain of the metavariable to a particular set of metacoded values. For instance, metavariable $X_1^3$ ranges over $\mathit{Values}[3+1+1]$ (the set of values metacoded five or more times). Although both, degree and elevation, are numerical attributes, degree is an absolute characteristics whereas elevation is a relative characteristic (it adjusts to the domain relative to degree). Thus, elevation is unchanged by metacoding and demetacoding (cf. Figure 5).

The positioning of a metavariable $X_h^n$ can be illustrated using a metasystem transition scheme as in Figure 7.

### 3.4 Configuration Values

The set of S-Graph-$n$ values is refined to form *configuration values* where, except for metavariables, all values are encoded at least once ($m \geq 1$). 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). Intuitively, one can think of configuration values $\mathit{cval}^n$ as a description language for characterizing metacoded data patterns occurring in a metasystem hierarchy at level $n$.[1] The importance of correctly abstracting from metacoded values, e.g. when generalizing across several levels of metacode, has

[1] Logic variables in queries play a similar role as metavariables in configuration values in that both are placeholders that specify unknown entities (note that the latter

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

**Fig. 8.** Configuration values

been explained elsewhere [28, 10]. Here we give a formal semantics that justifies the generalization and embedding operations defined in the following sections.

Formally, each configuration value $cval^n \in Cvalues[n]$ denotes a set $[\![cval^n]\!]^n \subseteq Values[n+1]$. Note that only $X_h^n$ are considered as metavariables on level $n$; metavariables with degree $n+m$, where $m \geq 1$, are treated as encoded constructors.

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

**Fig. 9.** The value sets denoted by configuration values ($m \geq 1$)

*Example 6.* Consider the configuration value $(\mathtt{CONS}^1\ (\mathtt{ATOM}^1\ \mathtt{a})\ Y_0^0)$ which, at reference level 0, represents a set $[\![(\mathtt{CONS}^1\ (\mathtt{ATOM}^1\ \mathtt{a})\ Y_0^0)]\!]^0 \in Values[1]$ of metacoded values:

$$\begin{aligned}
[\![(\mathtt{CONS}^1\ (\mathtt{ATOM}^1\ \mathtt{a})\ Y_0^0)]\!]^0 = \{&(\mathtt{CONS}^1\ (\mathtt{ATOM}^1\ \mathtt{a})\ (\mathtt{ATOM}^1\ \mathtt{a})), (\mathtt{CONS}^1\ (\mathtt{ATOM}^1\ \mathtt{a})\ (\mathtt{ATOM}^1\ \mathtt{b})), \ldots, \\
&(\mathtt{CONS}^1\ (\mathtt{ATOM}^1\ \mathtt{a})\ (\mathtt{ATOM}^2\ \mathtt{a})), (\mathtt{CONS}^1\ (\mathtt{ATOM}^1\ \mathtt{a})\ (\mathtt{ATOM}^2\ \mathtt{b})), \ldots, \\
&(\mathtt{CONS}^1\ (\mathtt{ATOM}^1\ \mathtt{a})\ (\mathtt{ATOM}^3\ \mathtt{a})), (\mathtt{CONS}^1\ (\mathtt{ATOM}^1\ \mathtt{a})\ (\mathtt{ATOM}^3\ \mathtt{b})), \ldots\}
\end{aligned}$$

Consider the configuration value again, but metacode it once. Now $(\mathtt{CONS}^2\ (\mathtt{ATOM}^2\ \mathtt{a})\ Y_0^1)$ represents a singleton set at reference level 0. This is justified because the metacoded value contains no longer metavariables that belong to reference level 0. The metavariable originally contained in the configuration value has been 'pushed down' one level due to metacoding.

$$[\![(\mathtt{CONS}^2\ (\mathtt{ATOM}^2\ \mathtt{a})\ Y_0^1)]\!]^0 = \{(\mathtt{CONS}^2\ (\mathtt{ATOM}^2\ \mathtt{a})\ Y_0^1)\}$$

---

are designed for multi-level metasystem hierarchies and that different metaprograms may perform different operations on configuration values, e.g. program specialization, inverse computation).

# 4 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.*, [28, 13]. 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 belonging to $cval_1^n$ or $cval_2^n$ is given by $[\![cval_1^n]\!]^n \cup [\![cval_2^n]\!]^n$. Using configuration values as a description language, the goal of generalization is to find a description $cval_g^n$ such that $[\![cval_1^n]\!]^n \cup [\![cval_2^n]\!]^n \subseteq [\![cval_g^n]\!]^n$. Moreover, the $[\![cval_g^n]\!]^n$ should be as small (precise) as possible. In other words, $cval_g^n$ should be general enough to describe (contain) the sets $[\![cval_1^n]\!]^n$ and $[\![cval_2^n]\!]^n$, but it should be as specific as possible (to minimize the loss of information).

These concepts can be formalized by extending familiar concepts of *most-specific generalization*, as applied previously to program specialization in *e.g.*, [22], to level-index constructors and elevated metavariables.

## 4.1 Elevation and Generalization

Not only the positioning of metavariables by their degree is important in a hierarchy of metasystems (cf. Example 6), but also the domain specified by the elevation index is crucial. Elevation can be seen as (dynamic) typing of variables positioned in a metasystem hierarchy. One might imagine enforcing well-formedness with a type-system. We have not pursued this option, since typed languages are notoriously hard to work with in self-applicable program specialization.

*Example 7.* Suppose we want to generalize component $(\mathtt{ATOM}^2\ \mathtt{a}) \in \mathit{Values}[2]$ in

$$(\mathtt{CONS}^2\ (\mathtt{ATOM}^2\ \mathtt{a})\ Y_0^1) \in \mathit{Values}[2]$$

such that at reference level 0 the resulting configuration value $cval^0$ describes a set $[\![cval^0]\!]^0 \subseteq \mathit{Values}[2]$. Introducing a metavariable $X_0^0$ leads to configuration value

$$[\![(\mathtt{CONS}^2\ X_0^0\ Y_0^1)]\!]^0 \not\subseteq \mathit{Values}[2]$$

In other words, the configuration value does not satisfy our condition because it admits values such as

$$(\mathtt{CONS}^2\ (\mathtt{ATOM}^1\ \mathtt{a})\ Y_0^1) \notin \mathit{Values}[2]$$

Without the use of elevation, a subset of $Values[2]$ cannot be described by $cval^0$. Adjusting the domain of the metavariable relative to the degree, $X_1^0$, gives the desired generalization.

$$[\![(\texttt{CONS}^2\ X_1^0\ Y_0^1)]\!]^0 \subseteq Values[2]$$

### 4.2 Most Specific Domain Index

The most specific domain index will help us to determine the domain of metavariables upon generalization. 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 $msdi^n(cval^n)$).

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

**Fig. 10.** Most specific domain index ($m \geq 1$)

*Example 8.* Consider the following examples of most specific domain indices.

$$msdi^0((\texttt{CONS}^2\ (\texttt{ATOM}^3\ \texttt{a})\ X_1^0)) = 1 \tag{4}$$
$$msdi^0((\texttt{CONS}^5\ (\texttt{ATOM}^3\ \texttt{a})\ X_1^3)) = 2 \tag{5}$$
$$msdi^3((\texttt{CONS}^5\ (\texttt{ATOM}^6\ \texttt{a})\ (\texttt{ATOM}^4\ \texttt{b}))) = 0 \tag{6}$$

### 4.3 Most Specific Generalization

We are now in the position to define most specific generalization. First we introduce elevation-compatible substitution and related operations.

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* $n$, $\theta^n = \{X_{i_1}^n{}_{h_1} := cval_1^n, \ldots, X_{i_t}^n{}_{h_t} := cval_t^n\}$, is a finite set of bindings such that the metavariables $X_{i_i}^n{}_{h_i}$ 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 $\theta^n$ is a substitution at level $n$.

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

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

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

*where* $Y, Z$ *are fresh and* $i = \min(msdi^n(v), msdi^n(v'))$, $j = \min(msdi^n(w), msdi^n(w'))$

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

**Fig. 11.** Computation of most specific generalization on level $n$ ($m \geq 1$)

**Definition 1 (renaming, generalization, msg).**

1. *A* renaming substitution *for* $cval^n$ *is a substitution* $\{X_{_1h_1}^{\ n} := Y_{_1h_1}^{\ n}, \ldots, X_{_th_t}^{\ n} := Y_{_th_t}^{\ n}\}$ *such that there is a binding for every metavariable occurring in* $cval^n$ *and* $Y_{_ih_i}^{\ n}$ *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* $\theta_1^n$ *and* $\theta_2^n$ *are elevation-compatible substitutions,* $cval_1^n \equiv cval_{gen}^n\theta_1^n$ *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* $cval_{gen}'^n$.

**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 11 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 (most specific generalization).** *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.* 1. *Termination of the algorithm.* In every rewrite rule the size of the substitutions $\Theta, \Theta'$ 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 by induction over the length of the reduction that the algorithm always computes a generalization.
   The initial triple is clearly a generalization. We will show one case; the rest is similar. Consider the third reduction rule. We will prove that the right hand side is a generalization of $cval_1^n$ and $cval_2^n$. Let $\theta''$ be $\{Y_i^n := v, Z_j^n := w\} \cup \theta$. We have:
   $$\begin{array}{ll} \tau\{X_h^n := (\texttt{CONS}^{n+m}\ Y_i^n\ Z_j^n)\}\theta'' & = \\ \tau\{X_h^n := (\texttt{CONS}^{n+m}\ Y_i^n\ Z_j^n)\theta''\}\theta'' & = \\ \tau\{X_h^n := (\texttt{CONS}^{n+m}\ v\ w)\}\theta'' & = \\ \tau\{X_h^n := (\texttt{CONS}^{n+m}\ v\ w)\}\theta & =_{\text{induction}} \\ cval_1^n & \end{array}$$
   The second last equality holds because $Y, Z$ are fresh variables. Analogously we can prove that:
   $$\tau\{X_h^n := (\texttt{CONS}^{n+m}\ Y_i^n\ Z_j^n)\}\{Y_i^n := v', Z_j^n := w'\} \cup \theta' = cval_2^n$$
   The condition on the rule ensures that the new substitutions are elevation-compatible, thus we have proved the case.
3. *Computation of msg.* We prove by contradiction that the generalization computed by the algorithm is an msg. Assume that the generalization $(cval_{gen}^n, \theta_1, \theta_2)$ computed by the algorithm is not an msg, then there exists an msg $(cval_{msg}^n, \theta_1', \theta_2')$ such that $cval_{msg}^n = cval_{gen}^n\theta^n$ where $\theta^n$ is not a renaming substitution. Therefore it must hold that $\theta^n\theta_1' = \theta_1$ and $\theta^n\theta_2' = \theta_2$. Since $\theta^n$ is not a renaming substitution it must have one of the two forms:
   $$\begin{array}{l} \{X_h^n := (\texttt{ATOM}^{n+m}\ atom)\} \cup \theta'^n \\ \{X_h^n := (\texttt{CONS}^{n+m}\ v\ w)\} \cup \theta'^n \end{array}$$
   for some substitution $\theta'^n$. But if $\theta^n$ has one of these forms also $\theta_1$ and $\theta_2$ will have that form and either the second or the third rule of Figure 11 would apply and $(cval_{gen}^n, \theta_1, \theta_2)$ would not be in normal form. So we have arrived at a contradiction and the generalization computed by the algorithm must therefore be most specific.

*Property 2 (metacode invariance of msg).* For any $cval_1^n, cval_2^n \in Cvalues[n]$, the most specific generalization at level $n$ is *invariant wrt. repeated metacoding* $\mu^m, m \geq 0$. Let
$$(cval^n, \theta_1^n, \theta_2^n) = msg^n(cval_1^n, cval_2^n)$$
for some $cval^n$, $\theta_1^n$ and $\theta_2^n$ then there exists $cval^{n+m}$, $\theta_3^n$ and $\theta_4^n$ such that
$$(cval^{n+m}, \theta_3^n, \theta_4^n) = msg^{n+m}(\mu^m\{cval_1^n\}, \mu^m\{cval_2^n\})$$
and $\mu^m\{cval^n\} = cval^{n+m}$ (modulo renaming of variables).

*Example 9.* Following are three examples of most specific generalization at level 2 and level 3. The first and the last illustrate the invariance property of msg:
$$\begin{array}{ll} msg^2((\texttt{CONS}^3\ X_2^2\ (\texttt{ATOM}^4\ \texttt{a})), (\texttt{CONS}^3\ Y_1^2\ (\texttt{ATOM}^3\ \texttt{a}))) = & ((\texttt{CONS}^3\ Z_1^2\ R_0^2), \\ & \{Z_1^2 := X_2^2, R_0^2 := (\texttt{ATOM}^4\ \texttt{a})\}, \\ & \{Z_1^2 := Y_1^2, R_0^2 := (\texttt{ATOM}^3\ \texttt{a})\}) \end{array}$$

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

$$msg^3((\mathtt{CONS}^4\ X_2^3\ (\mathtt{ATOM}^5\ \mathtt{a})), (\mathtt{CONS}^4\ Y_1^3\ (\mathtt{ATOM}^4\ \mathtt{a}))) = ((\mathtt{CONS}^4\ Z_1^3\ R_0^3), \{Z_1^3 := X_2^3, R_0^3 := (\mathtt{ATOM}^5\ \mathtt{a})\}, \{Z_1^3 := Y_1^3, R_0^3 := (\mathtt{ATOM}^4\ \mathtt{a})\})$$

## 5 Embedding of Configuration Values

The homeomorphic embedding relation known from term algebra [8] 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 [19, 22, 26].

In our setting, the embedding relation works on specialization states described by configuration values.

**Definition 3 (Homeomorphic embedding).** *The homeomorphic embedding relation $\trianglelefteq^n \subseteq Cvalues[n] \times Cvalues[n]$ is the smallest relation satisfying the rules in Figure 12.*

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. Because there may be infinitely many variables we have to be conservative and let all encoded variables embed all other encoded variables (as long as these have the same index and elevation). 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 $cval_1^n \trianglelefteq^n cval_2^n$.

The homeomorphic embedding relation as it is defined is, however, not a well quasi-ordering (WQO) on $Cvalues[n]$, since the index and the elevation of of elements in $Cvalues[n]$ is not limited, but since we assume that we always work with a finite number of elevations and therefore also maximum encoding we have the following theorem which is sufficient for ensuring termination for MST schemes:

**Theorem 2 (WQO $\trianglelefteq^n$).** *The relation $\trianglelefteq^n$ is a well-quasi order on $Cvalues[n] \backslash (Cvalues[n+m] \cup \{X_h^n \mid h \geq m\})$.*

*Proof.* Proven by a simple adaption of the proof in the literature [8], since there are only finitely many constructors in $Cvalues[n] \backslash (Cvalues[n+m] \cup \{X_h^n \mid h \geq m\})$[2]

[2] Without the limitation on the domain $Cvalues[n]$ we could have infinitely many constructors, e.g. $(\mathtt{ATOM}^{n+m}\ \mathtt{a}), m \geq 1$.

$$X_h^{n+r} \trianglelefteq^n Y_h^{n+r} \qquad [VAR]$$

$$(\mathtt{ATOM}^{n+m}\ atom) \trianglelefteq^n (\mathtt{ATOM}^{n+m}\ atom) \qquad [ATOM]$$

$$\frac{cval_1 \trianglelefteq^n cval_1' \qquad cval_2 \trianglelefteq^n cval_2'}{(\mathtt{CONS}^{n+m}\ cval_1\ cval_2) \trianglelefteq^n (\mathtt{CONS}^{n+m}\ cval_1'\ cval_2')} \qquad [CONS]$$

$$\frac{cval \trianglelefteq^n cval_1}{cval \trianglelefteq^n (\mathtt{CONS}^{n+m}\ cval_1\ cval_2)} \qquad [DIVE\text{-}L]$$

$$\frac{cval \trianglelefteq^n cval_2}{cval \trianglelefteq^n (\mathtt{CONS}^{n+m}\ cval_1\ cval_2)} \qquad [DIVE\text{-}R]$$

**Fig. 12.** Homeomorphic embedding on level $n$ ($m \geq 1,\ r \geq 0$)

*Property 3 (metacode invariance $\trianglelefteq^n$).* For any $cval_1^n, cval_2^n \in Cvalues[n]$, relation $\trianglelefteq^n$ is *invariant wrt. repeated metacoding* $\mu^m, m \geq 0,\ k \geq 0$:

$$cval_1^n \trianglelefteq^n cval_2^n \iff \mu^m\{cval_1^n\} \trianglelefteq^{n+k} \mu^m\{cval_2^n\}$$

In other words, the invariance of $\trianglelefteq^n$ wrt. repeated metacoding avoids the problems described in Example 2.

*Example 10.* Here are a few examples with the homeomorphic embedding relation:

$$(\mathtt{ATOM}^3\ \mathtt{a}) \trianglelefteq^2 (\mathtt{CONS}^3\ (\mathtt{ATOM}^3\ \mathtt{a})\ X_1^2)$$
$$(\mathtt{ATOM}^3\ \mathtt{a}) \not\trianglelefteq^2 (\mathtt{CONS}^3\ (\mathtt{ATOM}^4\ \mathtt{a})\ X_1^2)$$
$$Y_0^2 \trianglelefteq^2 (\mathtt{CONS}^3\ (\mathtt{ATOM}^4\ \mathtt{a})\ X_0^2)$$
$$Y_1^2 \not\trianglelefteq^1 (\mathtt{CONS}^3\ (\mathtt{ATOM}^4\ \mathtt{a})\ X_0^2)$$

## 6 Self-Applicable Specialization

The previous sections were concerned with the development of the foundations and 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 13 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).

*Initial call to the (self-applicable) specializer:*

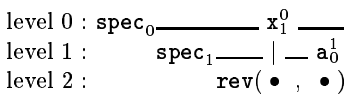
```
(let tree (call-1 (rev (cons-1 (atom-1 1)
                               (cons-1 (atom-1 2) (atom-1 nil)))
                       (mv 0 a)))
  (let defs (cons (cons (atom rev) (atom nil))
                  (cons <rev metacoded once> (atom nil)))
    (call (spec-start tree defs)))))
```

*Result of specialization:*

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

**Fig. 13.** Specialization example

*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 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:




We now examine some essential steps in the development of the computation. The initial call of metasystem hierarchy is shown in Figure 14. 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

*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))))
              ...)
      ...))
```

**Fig. 14.** Specialization example showing embedding and generalization

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 14. 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 12, which means that we consider:

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

instead. The first of these two clearly holds because of the *VAR*-rule in Figure 12. 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 specializer in the example replaces the early configuration by the most specific generalization of the two configurations. For our example Figure 14 shows the msg of `(1)` and `(2)` that is obtained by the the method shown in Figure 11.

## 7 Related Work

The ideas presented in this paper have been heavily influenced by three concepts present in Turchin's work [25]: metacoding, metavariables, and metasystem transition. Subsequently, these concepts have been formalized [10] and studied in different contexts, *e.g.* [13, 26].

Representing and reasoning about object level theories is an important field in logic and artificial intelligence (e.g. different encodings have been discussed in [14]) and has led to the development of logic languages that support declarative metaprogramming (*e.g.* the programming language Gödel [15]). Logic variables and unification as provided by their underlying logic system, lack, among others, the notion of elevation and the direct support for multiply encoded data. In recent work [13], we therefore proposed a simple language S-graph-$n$ to study meta-programming aspects of self-applicable online program specialization. It will be interesting to study and possibly incorporate some of the notions developed here in a logic programming context (e.g. by extending an existing system).

MetaML, a statically typed multi-level language for hand-writing multi-level generating extensions was introduced in [24]. Another multi-level programming language is *Alloy* [4], 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 [11] 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 [9].

Most specific generalization and the homeomorphic embedding relation are known from term algebra [8]. Variants of this relation are used in termination proofs for term rewrite systems and for ensuring local termination of partial deduction [5]. After it was taken up in [22], it has inspired more recent work [2, 19, 27, 26].

## 8 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.

On the theoretical side a thorough comparison of the approach taken in this contribution with higher-order logic programming concepts is on the agenda. Another challenging question is whether such approaches can be combined and used for metaprogramming in general.

To conclude, we believe the work presented in this paper is a solid basis for future work on multi-level program hierarchies and their efficient implementation on the computer.

**Acknowledgements**

We would like to thank Michael Leuschel and Bern Martens for stimulating discussions on topics related to this work. Anonymous referees as well as participants of LOPSTR'98 provided very useful feedback on this paper.

## References

1. S.M. Abramov. Metacomputation and program testing. In *1st International Workshop on Automated and Algorithmic Debugging*, pages 121–135, Linköping University, Sweden, 1993.
2. 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.
3. K. Apt and F. Turini *Meta-Logics and Logic Programming*, MIT Press, 1995.
4. J. Barklund. A basis for a multilevel metalogic programming language. Technical Report 81, Uppsala University, Dept. of Computing Science, 1994.
5. R. Bol. Loop checking in partial deduction. *J of Logic Programming*, 16(1&2):25–46, 1993.
6. S. Costantini and G. Lanzarone. A metalogic programming language. In G. Levi and M. Martelli, editors, *Proceedings Sixth International Conference on Logic Programming*, pages 218–233. MIT Press, 1989.
7. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
8. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 244–320. Elsevier, 1992.
9. 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.
10. 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.

11. R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.
12. R. Glück and A.V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, et al., editors, *Static Analysis. Proceedings.* Lecture Notes in Computer Science, Vol. 724, pages 112–123. Springer-Verlag, 1993.
13. J. Hatcliff and R. Glück. Reasoning about hierarchies of online specialization systems. In Danvy et al. [7].
14. P. Hill and J. Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, 1994.
15. P. Hill and J.W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.
16. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
17. M. Leuschel. Homeomorphic embedding for online termination. Technical Report DSSE-TR-98-11, University of Southampton, Dept. of Electronics and Computer Science, 1998.
18. M. Leuschel. On the power of homeomorphic embedding for online termination. G. Levi, editor, *Static Analysis. Proceedings.* Lecture Notes in Computer Science, Vol. 1503, pages 230–245, Springer-Verlag 1998.
19. M. Leuschel and B. Martens. Global control for partial deduction through characteristic atoms and global trees. In Danvy et al. [7], pages 263–283.
20. G. Nadathur and D. Miller. Higher-Order Logic Programming. In Handbook of Logic in AI and Logic Programming, Vol. 5, pages 499–590, Oxford University Press, 1998.
21. A. Pettorossi, M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19 & 20:261–320, 1994.
22. 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.
23. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.
24. 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.
25. V.F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
26. V.F. Turchin. Metacomputation: metasystem transitions plus supercompilation. In Danvy et al. [7].
27. 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.
28. 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.
29. W. Vanhoof and B. Martens. To parse or not to parse. In N. Fuchs (ed.), *Logic Program Synthesis and Transformation (LOPSTR'97)*, Lecture Notes in Computer Science, Vol. 1463, pages 314–333, Springer-Verlag 1998.