Reasoning about Hierarchies
of Online Program Specialization Systems *

John Hatcliff
Robert Glück

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

Abstract. We present the language S-Graph-n — the core of a multi-
level metaprogramming environment for exploring foundational issues of
self-applicable online program specialization.

We illustrate how special-purpose S-Graph-n primitives can be used
to obtain an efficient and conceptually simple encoding of programs as
data objects. The key feature of the encoding scheme is the use of nu-
merical indices which indicate the number of times that a program piece
has been encoded.

Evaluation of S-Graph-n is formalized via an operational seman-
tics. This semantics is used to justify the fundamental operations on
metavariables — special-purpose tags for tracking unknown values in
self-applicable online specialization systems. We show how metavari-
ables can be used to construct biased generating extensions without relying on
a separate binding-time analysis phase.

1 Introduction

Metasystem hierarchies have been used for more than a decade to generate com-
ponents and other program generators. A metasystem hierarchy is any situation
where a program $p_0$ is manipulating (e.g., interpreting, compiling, transform-
ing) another program $p_1$. Program $p_1$ may be manipulating another program $p_2$,
and so on. A metasystem hierarchy can be diagrammed using an Metasystem
Transition (MST) scheme as in Figure 1 [7,9,27].

The best known examples are the Futamura projections which were the driving
force behind the initial work on self-application of program specialization
systems. This work identified binding-time analysis as a useful tool for attacking
the fundamental problem of tracking unknown values, and regarded associated
offline specialization as essential for taming self-application [15,16].

On the other hand, more powerful online specialization methods, such as super-
compilation and partial deduction have not yet given satisfactory results for all
Futamura projections (not to mention multiple self-application or the special-
izer projections). It seems harder to reason about the behavior of hierarchies of
online systems for several reasons.

1. Semantics of multi-level specialization: There is no static staging of
programs (as binding-time analysis gives in offline systems). This makes
it harder to predict the behavior of the specializer.

* This work has been supported by the Danish Research Academy and by the DART
project (Design, Analysis and Reasoning about Tools) of the Danish Research
Councils.
II. Identifying positions in hierarchies: There is no analogue to the typing information given by binding-time analysis — that is, it is harder to determine at which level in the hierarchy an entity resides. This is complicated in cases where the hierarchy contains many layers. Multiple encodings of programs as data objects can cause exponential growth in size and further obscure hierarchical positions of objects.

III. Tracking unknowns: It is unclear how to establish semantic principles for tracking unknown values across multiple levels in the metasystem hierarchy. This makes it more difficult to design e.g., a self-applicable online partial evaluator that can produce "biased" generating extensions and satisfy the "Mix criteria" (i.e., all binding-time tests are reduced at specialization time) [14, Chapter 7].

IV. Experimentation: There are simply fewer existing systems for performing experiments. Moreover, existing systems often lack proper metaprogramming environments.

Our goal is to identify and clarify the foundational issues involved in hierarchies of online specialization systems. We believe the best way to achieve this goal is to develop a very simple online specialization system which focuses tightly on the problematic points of online specialization noted above: (I) semantics of specialization, (II) properties of program encodings and identifying position of entities in a hierarchy, (III) tracking unknown values across levels in metasystem hierarchies. The system should also provide a metaprogramming environment which allows one to easily construct different specialization systems (IV). Moreover, the system should be well-suited for supporting stronger forms of online specialization such as supercompilation.

Outline

In this paper, we report on the initial design and partial implementation of such a system. The system is based on S-Graph — a very simple language which has been used to study the foundations of supercompilation [9] and neighborhood analysis [1]. Section 2 revisits the syntax and semantics of S-Graph.

Section 3 discusses problems of program encodings. Based on this discussion, Section 4 presents a new version of S-Graph called S-Graph-n which contains language primitives especially designed for manipulating metacode — data objects representing programs.

Section 5 presents special S-Graph-n primitives for representing metavariables — tags for tracking unknown values across multiple levels in the hierarchy.
Building on previous work [7, 27], we formalize semantics of metavariables and illustrate how they can be used to construct a simple self-applicable online partial evaluator which can produce “biased” generating extensions without relying on a separate binding-time analysis phase.

Section 6 discusses related work, and Section 7 concludes.

## 2 S-Graph

Figure 2 presents the syntax of S-Graph — a first-order, functional programming language restricted to tail-recursion. As the name implies, one can think of S-Graph programs as being textual representations of graphs. The only data objects are well-founded, i.e., non-circular, S-expressions (as known from Lisp). 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, constructors CONS, program variables (PV name), and atomic constants (drawn from an infinite set of symbols).

Note the conditional in S-Graph: the test cntr may update the environment.

As in supercompilation, we refer to such tests as contractions [23]. Two elementary contractions are sufficient for S-expressions:

- (EQA? arg1 arg2) — tests the equality of two atoms denoted by arg1 and arg2. If the arguments are non-atomic then the test is undefined.
- (CONS? exp h t) — if the value of exp is a pair (CONS val1 val2), then the test succeeds and the variable h is bound to val1 and the variable t to val2; otherwise, the test fails.

The arguments of function calls and contractions are restricted to variables and atomic constants in order to limit the number of places where values may be constructed. Because there are no nested function calls, we describe the language as flat (i.e., it corresponds to a flow-chart language). Syntactic sugar: we write 'atom as shorthand for (ATOM atom); lower case identifiers as shorthand for (PV name). Figure 3 shows the list reverse written in S-Graph.

## 3 Programs as Data Objects

Metaprogramming requires facilities for encoding programs as data objects. In general, metaprogramming also requires encoding tags (which we call metavari-
(DEFINE (REVERSE x)
  (CALL (LOOP x 'NIL)))

(DEFINE (LOOP x bag)
  (IF (CUN$? x head tail)
      (LET headbag (CUN$ head bag)
                (CALL (LOOP tail headbag))
         bag))

Fig. 3. List reverse in S-Graph

ables) for representing unknown entities. The encoding must be injective. This
ensures that all objects are encoded uniquely, and that encoded objects can be
“recovered” with a decoding. For historical reasons, we refer to the encoding as
metacoding and to decoding as demetacoding [26].

In a metasystem hierarchy such as displayed in Figure 1, programs and
metavars are metacoded many times. For example, pn of Figure 1 must be
metacoded (directly or indirectly) n times, since n systems lie above it in the
hierarchy. The number of encodings is called degree [27]. As the discussion of pn
illustrates, an object’s degree indicates its vertical position in a hierarchy.

In general, there may be different languages with different metacodings on
each level of a hierarchy. For simplicity, we consider only one language and
assume that the same metacode is used on all levels.

3.1 Metasystem Hierarchies

Metasystem hierarchies are a cornerstone of Turchin’s approach: the construc-
tion of hierarchies of arbitrary height is taken as the basis for program analy-
sis and transformation [21] (in contrast to logics and mathematics which usually
deal with two-level hierarchies). For example, the well-known Futamura pro-
jections make use of a three-level hierarchy of metasystems (i.e., program special-
izers). To make multi-level hierarchies practical, one needs facilities for satisfying
the following requirements.

1. Efficient encoding: low space consumption and minimal overhead for ma-
nipulating the encoding. This is essential because repeated metacodings in a
metasystem hierarchy may require a significant amount of space and processing
time. A straightforward encoding may lead to growth that is exponential
in the number of metacodings.

2. Efficient computation on all levels of a metasystem hierarchy. Generally
speaking, the higher the hierarchy, the slower the overall transforma-
tion. Since most non-trivial self-applicable specializers incorporate a self-
interpreter for evaluating static expressions, the run-time of multiple self-
application typically grows exponentially with the number of self-application
levels.

These problems can dramatically impact usability, e.g., of self-application [6,8].
Various approaches have been employed to improve the performance and the
manipulation of representations. For instance, the logic programming language
Turchin's freezer mechanism [24] facilitates a dramatic optimization in run time during multiple self-applications [17]. However, it remains to extend and systematically clarify similar techniques in the context of multi-level hierarchies.

Our goal is to lay the foundation for a multi-level metaprogramming environment that fully addresses requirements (1) and (2) above, and that is not biased towards a particular method for metacomputation (e.g., partial evaluation, supercompilation). Meeting each of the above requirements depends on the strategies used to (i) represent programs as data (i.e., metacoding), (ii) represent computed values, and (iii) represent known/unknown entities. These representations should be cheap (wrt space), eliminable (i.e., removable by specialization), and efficient to process at specialization time. In the present work, we concentrate on requirement (1) as a prerequisite for (2).

### 3.2 Criteria for Metacoding

**Low space consumption.** To illustrate the space consumption problem, consider the straightforward strategy for metacoding S-Graph programs given in Figure 4. The encoding is simple, but leads to an exponential growth of expressions. Using this strategy, the S-Graph expression `(CONS (ATOM a) (ATOM b))` metacoded once is

\[
\mu((\text{IF } \text{cntr } \text{tree}_1 \text{ tree}_2)) = (\text{CONS} (\text{ATOM IF}) (\text{CONS} \mu(\text{cntr}) (\text{CONS} \mu(\text{tree}_1) \mu(\text{tree}_2))))
\]

\[
\mu((\text{LET name exp } \text{tree})) = (\text{CONS} (\text{ATOM LET}) (\text{CONS} (\text{ATOM name}) (\text{CONS} \mu(\text{exp}) \mu(\text{tree}))))
\]

... 

\[
\mu((\text{CONS } \text{exp}_1 \text{ exp}_2)) = (\text{CONS} (\text{ATOM CONS}) (\text{CONS} \mu(\text{exp}_1) \mu(\text{exp}_2)))
\]

\[
\mu((\text{ATOM atom})) = (\text{CONS} (\text{ATOM ATOM}) (\text{ATOM atom}))
\]

**Fig. 4. S-Graph metacoding (excerpts)**

and metacoded twice is

\[
\mu((\text{CONS ATOM ATOM))) = (\text{CONS} (\text{ATOM CONS}) (\text{CONS} (\text{CONS} (\text{ATOM ATOM}) (\text{ATOM a}))) (\text{CONS} (\text{ATOM CONS}) (\text{CONS} (\text{CONS} (\text{ATOM ATOM}) (\text{ATOM b}))) (\text{CONS} (\text{ATOM CONS}) (\text{CONS} (\text{CONS} (\text{CONS} (\text{ATOM ATOM}) (\text{ATOM atom})))) (\text{CONS} (\text{ATOM CONS}) (\text{CONS} (\text{CONS} (\text{CONS} (\text{ATOM ATOM}) (\text{ATOM atom}))) (\text{CONS} (\text{ATOM CONS}) (\text{CONS} (\text{CONS} (\text{CONS} (\text{ATOM ATOM}) (\text{ATOM b}))))))))
\]

Each overbar represents an application of the metacoding function.

**Efficient to process.** The encoding must enable facilities for efficient computation on any level in a metasystem hierarchy. For example, the straightforward encoding shown above is not very efficient to process: the time to access a sub-component (e.g., `(ATOM b)`) in the metacode grows with each level of encoding. Moreover, the time to metacode expressions grows exponentially with each level of encoding.
Compositionality. One often needs to embed data with lower degrees as components of constructs with higher degrees (e.g., in a representation of partially static structures). We describe such structures as non-monotonic with respect to degree. In some popular metacoding strategies, changing the degree of a single subcomponent may require a non-local modification of the data structure. A familiar example is the quote mechanism à la Scheme. Consider using this strategy to metacode the expression \((\text{CONS} \ (\text{ATOM} \ a) \ (\text{ATOM} \ b))\) two times.

\[
\text{CONS} \ (\text{ATOM} a) \ (\text{ATOM} b) = \text{CONS} \ (\text{CONS} \ (\text{ATOM} a) \ (\text{ATOM} b))
\]

Suppose we wish to replace the subexpression \((\text{ATOM} a)\) by a variable (representing an unknown value). This requires a non-local change. We have to modify the enclosing expression by “pushing quotes inside” and using functions (here \text{list} to rebuild the enclosing data structure.

\[
\text{CONS} \ a \ (\text{ATOM} b) = \text{LIST} \ \text{CONS} \ a \ (\text{ATOM} b)
\]

A similar situation occurs using the backquote/comma mechanism. Representing multiply encoded structures with several different degrees of “unknowns” is even more cumbersome using this metacoding strategy.

Conceptually simple and easy to reason about. In metasystem hierarchies, one often needs to reason about the degree and position of data in a hierarchy, or simply trace the computation in a metasystem hierarchy. The \texttt{S-Graph} encoding of Figure 4 clearly does not satisfy this criteria. Non-compositional encodings such as the quote mechanism are also somewhat unsatisfactory in this regard since it is not possible to reason locally. For example, one cannot determine the degree of a subcomponent without consulting the entire enclosing expression. The degree of a subcomponent is relative to the degree of the enclosing expression.

We now summarize the advantages and disadvantages of the quote/backquote mechanism and motivate our proposed solution.

**Quote/Backquote Mechanism:** The advantages of using the quote/backquote mechanism include low space consumption, and fast encoding and decoding (simply adding/removing a quote around an \texttt{S-expression}). The time to encode/decode is constant and independent of the number of levels. The disadvantage of is that changing the degree of a single subexpression may require a non-local modification of the data structure, as shown above. Moreover, one cannot reason locally about a subexpression because the encoding is relative, that is, the degree of a subexpression depends on the degree of the enclosing expression. The quote/backquote mechanism is an appropriate solution if a monotone, relative, and non-compositional encoding is sufficient. However, this is generally not the case in metasystem hierarchies.

**Level Indexing:** In the following section, we introduce language primitives that rely on numerical indices to indicate the number of program encodings. Using the level-indexed primitives, the above expression metacoded once is

\[
\text{CONS} \ (\text{ATOM} a) \ (\text{ATOM} b) = \text{CONS} \ (\text{CONS} \ (\text{ATOM} a) \ (\text{ATOM} b)) \ (\text{ATOM} a)
\]

and metacoded twice is
We believe that level indices are preferable for several reasons. Metacoding using
level-indexing is space efficient, compositional and allows non-monotonic
structures. Also, the typing intuition given by level indices is preferable from
a conceptual standpoint; the degree (and thus the vertical position in a hier-
archy) can be determined immediately using only local reasoning. Level-indexing
also facilitates metasystem jumps [17] where control is transferred between levels
in a hierarchy. This technique can drastically reduce the second problem of
metasystem hierarchies noted above: computation time.

A disadvantage of level-indexing is that metacoding requires time that is
linear in the size of the expression (as opposed to constant time for quote).
However, this is not a big factor in transformation time since metacoding is
usually performed in a pre-processing phase. In situations where metacoding and
demetacoding do occur during transformation (e.g., when reifying and reflecting
data), this cost can be minimalized by parameterizing primitive operations wrt
degree. This will be the essence of our approach to implementing metasystem
jumps.

4 S-Graph-$n$

4.1 Syntax

Figure 5 presents the syntax of S-Graph-$n$ — a language with special-purpose
constructs for representing programs as data, and tracking unknowns. S-Graph-$n$
trees are constructed from objects of the set $Sgn$ — level-indexed abstract
syntax tree nodes. For example, the following are S-Graph-$n$ components.

\[
\begin{align*}
(\text{IF}^2 (\text{CONS}^1 \text{ATOM}^3 \text{foo}) (\text{ATOM}^1 \text{bar})) (\text{PV}^3 x) (\text{PV}^4 y)) & \quad (1) \\
(\text{IF}^2 (\text{EQ}^2 \text{ATOM}^1 \text{foo}) (\text{ATOM}^2 \text{bar})) (\text{PV}^3 x) (\text{PV}^4 y)) & \quad (2) \\
(\text{IF}^3 (\text{EQA}^2 \text{ATOM}^3 \text{foo}) (\text{ATOM}^1 \text{bar})) (\text{CONS}^1 (\text{ATOM}^2 1) (\text{ATOM}^2 1)) (\text{PV}^1 y)) & \quad (3)
\end{align*}
\]

Intuitively, the S-Graph-$n$ components are indexed building blocks for construct-
ing multiply metacoded program pieces. Components can be composed in a
fairly arbitrary manner. This allows non-monotonic encodings (as motivated in
the previous section). Line (1) illustrates that one can also build components
that do not correspond to well-formed programs pieces. This is a common situa-
tion in metaprogramming applications: well-formedness must be enforced by
the programmer.²

The *index* of an S-Graph-$n$ component $sng$ is the level number attached to the
outermost construct of $sng$. In the components above, the index of (1) and
(2) is 2, and the index of (3) is 3.

The *degree* of an S-Graph-$n$ component $sng$ (denoted $\text{degree}(sng)$) is the small-
est index occurring in $sng$. In the components above, (1) and (3) have degree
1; (2) 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

² One might imagine enforcing well-formedness with a type system. We do not pursue
this option here, since typed languages are notoriously hard to work with in self-
applicable program specialization.
Components:
\[\text{sgn} \in \text{Sgn}\]
\[\text{sgn} ::= \begin{array}{l}
\text{(IF} \text{ sgn sgn sgn)} | \text{(LET} \text{ name sgn sgn)} | \ldots | \\
\text{(CONS} \text{ sgn name name)} | \text{(EQA} \text{ name sgn name)} | \\
\text{(MC} \text{ sgn name name name)} | \text{(MV} \text{ name name sgn)} | \ldots | \\
\text{(CONS} \text{ sgn sgn)} | \text{(ATOM} \text{ atom)} | \text{(PV} \text{ name)} | \\
\text{(MC} \text{ sgn sgn sgn)} | \text{(MV} \text{ h name}) \quad \text{for any} \ n, b \geq 0
\end{array}\]

Trees at level \(n\):
\[\text{tree}^n \in \text{Tree}[n]\]
\[\text{tree}^n ::= \begin{array}{l}
\text{(IF} \text{ cnt} \text{ tree}^n \text{ tree}^n) | \text{(LET} \text{ name exp} \text{ tree}^n) | \\
\text{(CALL} \text{ (fname arg')} | \text{ exp'})
\end{array}\]
\[\text{cnt}^n \in \text{Cnt}[n]\]
\[\text{cnt}^n ::= \begin{array}{l}
\text{(CONS} \text{ sgn name name name)} | \text{(EQA} \text{ arg'} \text{ arg'') | \\
\text{(MC} \text{ arg'} \text{ name name name}) | \text{(MV} \text{ arg'} \text{ name name name})
\end{array}\]
\[\text{exp}^n \in \text{Exp}[n]\]
\[\text{exp}^n ::= \begin{array}{l}
\text{(CONS} \text{ exp} \text{ exp}) | \text{(MC} \text{ arg'} \text{ arg'} \text{ arg'') | arg'}) | \text{ mc}^n
\end{array}\]
\[\text{arg}^n \in \text{Arg}[n]\]
\[\text{arg}^n ::= \begin{array}{l}
\text{(ATOM} \text{ atom)} | \text{(PV} \text{ name)} | \text{(MV} \text{ h name}) \quad \text{for any} \ h \geq 0
\end{array}\]

Metacode at level \(n\):
\[\text{mc}^n \in \text{Metacode}[n]\]
\[\text{mc}^n ::= \begin{array}{l}
\text{(IF} \text{ exp} \text{ exp} \text{ exp}) | \text{(LET} \text{ name exp} \text{ exp}) | \ldots | \\
\text{(CONS} \text{ exp} \text{ exp}) | \text{(ATOM} \text{ exp} \text{ atom)} | \text{(PV} \text{ exp} \text{ name}) | \\
\text{(MC} \text{ exp} \text{ exp} \text{ exp}) | \text{(MV} \text{ exp} \text{ h name}) \quad \text{for any} \ m \geq 1, b \geq 0
\end{array}\]

Fig. 5. Syntax of S-Graph-n (excerpts)

been metacoded more times). As motivated in Section 3, degree indicates to which level of a hierarchy a component belongs.

A S-Graph-n component \text{sgn} is monotone if the indices of subcomponents are the same or are increasing as one descends down the abstract syntax tree of \text{sgn}. Formally, all leaf components are monotone; a non-leaf component \text{sgn} is monotone if all its immediate subcomponents are monotone, and all indices of immediate subcomponents are greater than or equal to the index of \text{sgn}. In the components above, (2) is monotone; (1) and (3) are not. Given a set \(S \subseteq \text{Sgn}\), \text{monotone}(S) denotes the subset of monotone components of \(S\).

Elements of \text{Tree}[n] \subseteq \text{Sgn} are well-formed trees at level \(n\). In the components above, (2) is a well-formed tree at level 2; (1) and (3) are not. A similar intuition lies behind the other syntactic categories of Figure 5.

Elements of \text{Metacode}[n] \subseteq \text{Tree}[n] represent encoded program components at level \(n\). Intuitively, these are pieces of abstract syntax trees (encoded \(m\) times) manipulated by the program (e.g., interpreter, specializer) running at level \(n\). In the components above, (1),(2), and (3) are all elements of \text{Metacode}[0]. However, only (2) is an element of \text{Metacode}[1] since \text{(ATOM bar)} in (1) and
\( \text{val}^n \in \text{Values}[n] \)
\( \text{val}^n ::= (\text{ATOM}^n \ \text{atom}) \mid (\text{MV}^n \ h \ \text{name}) \mid (\text{CONS}^n \ \text{val}^n \ \text{val}^n) \mid \\
(\text{IF}^{n+m} \ \text{val}^n \ \text{val}^n \ \text{val}^n) \mid (\text{LET}^{n+m} \ h \ \text{name} \ \text{val}^n \ \text{val}^n) \mid \ldots \mid \\
(\text{MC}^{n+m} \ \text{val}^n \ \text{val}^n \ \text{val}^n) \mid (\text{MV}^{n+m} \ h \ \text{name}) \)

for any \( h \geq 0, m \geq 1 \)

Fig. 6. S-Graph-\( n \) values (excerpts)

\( (\text{CONS}^1 \ (\text{ATOM}^2 \ 1) \ (\text{ATOM}^2 \ 1)) \) in (3) represent an atom and a CONS instruction in the program running at level 1 (i.e., they are not encoded program pieces relative to level 1).\(^3\)

The constructs \( (\text{MC}^n \ \text{any}^n \ \text{any}^n \ \text{any}^n) \), \( (\text{MC}^{n+m} \ \text{any}^n \ \text{name} \ \text{name}) \) are added to construct and destruct metacode. Metavars \( (\text{MV}^n \ h \ \text{name}) \) are added to track unknown through a hierarchy of specialization systems. The numerical index \( h \) is the elevation of the metavariable. The semantics of these constructs will be given in Section 4.3.

Although the semantics is given later, we can present the canonical terms (i.e., results) of evaluation \( \text{Values}[n] \subseteq \text{Exp}[n] \) (Figure 6). Intuitively, values at level \( n \) are either atoms, metavars, or CONS-cells at level \( n \), or metacode at level \( n \) (representing encoded programs from higher levels).

The S-Graph-\( n \) machine is implemented in Scheme and is parameterized by a reference level \( n \). Given a reference level \( n \), the machine will run programs which are well-formed at level \( n \). S-Graph-\( n \) components are represented using Scheme vectors. All data structures (e.g., environment, definition list, etc.) in the implementation are built using S-Graph-\( n \) components. This makes it trivial to rify and reflect data (i.e., to move objects up and down the hierarchy) — one need only adjust index values. Furthermore, since the machine runs relative to a certain level \( n \), passing control between various levels is trivial — one need only adjust the reference level. We do not take full advantage of this functionality in the present work; it is the foundation for a future investigation of metasystem jumps [27].

When programming in S-Graph-\( n \), one usually takes \( 0 \) as the reference level. In the programming examples that we give later, components where indices are omitted are at level \( 0 \).

4.2 Metacoding

Figure 7 gives the metacoding function \( \mu \) for S-Graph-\( n \). There is no increase in the size of the encoded program with each level of encoding — only indices are incremented.\(^4\)

---

\(^3\) We have omitted metacode components for DEFINE constructs for efficiency reasons. A list of definitions is represented instead by a list of function names and a list of function bodies. We have included metacode components for all other constructs because this gives a uniform representation of program trees independent of level. This facilitates an implementation of metasystem jumps.

\(^4\) The increase is logarithmic if one considers the number of bits needed to represent level indices. We ignore this since in practice, the number of encodings never exceeds standard word capacities.
\[
\begin{align*}
\mu(\text{IF}^n \text{ sgn}_1 \text{ sgn}_2 \text{ sgn}_3) &= \text{IF}^{n+1} \mu(\text{sgn}_1) \mu(\text{sgn}_2) \mu(\text{sgn}_3) \\
\mu(\text{LET}^n \text{ name} \text{ sgn}_1 \text{ sgn}_2) &= \text{LET}^{n+1} \mu(\text{sgn}_1) \mu(\text{sgn}_2) \\
\mu(\text{CONS}^n \text{ sgn}_1 \text{ sgn}_2) &= \text{CONS}^{n+1} \mu(\text{sgn}_1) \mu(\text{sgn}_2) \\
\mu(\text{ATOM}^n \text{ atom}) &= \text{ATOM}^{n+1} \text{ atom} \\
\mu(\text{MV}^n \text{ h name}) &= \text{MV}^{n+1} \text{ h name}
\end{align*}
\]

Fig. 7. Metacoding function for S-Graph-n (excerpts)

The following property gives characteristics of \( \mu \). \( \mu^n \) denotes the iterated application of \( \mu \) (i.e., \( \mu^n \{\text{sgn}\} \) is \( \text{sgn} \) metacoded \( n \) times). For any \( S \subseteq \text{Sgn} \), \( \mu \{S\} \) denotes the set obtained by element-wise application of \( \mu \).

**Property 1 (Properties of \( \mu \))**

1. **injectivity:** \( \forall \text{sgn}_1, \text{sgn}_2 \in \text{Sgn} \). \( \text{sgn}_1 \neq \text{sgn}_2 \Rightarrow \mu(\text{sgn}_1) \neq \mu(\text{sgn}_2) \)

2. **left inverse:** \( \forall \text{sgn} \in \text{Sgn} \). \( \text{sgn} = \mu_{-1}(\mu(\text{sgn})) \)

3. **embedding:** \( \forall n \geq 0 \). \( \mu^n(\text{Sgn}) \supset \mu^{n+1}(\text{Sgn}) \)

4. **canonical representation:**
   \( \forall n \geq 0 \). \( \mu^{n+1}(\text{Sgn}) \subseteq (\text{Metacode}[n] \cap \text{Values}[n]) \)

5. **preservation of syntactic categories:** \( \forall n \geq 0 \). \( \mu(C[n]) = C[n+1] \)
   where \( C \in \{\text{Tree, Exp, Arg, Cntx, Metacode, Values}\} \)

6. **preservation of monotonicity:** \( \mu\{\text{monotone}(\text{Sgn})\} \subseteq \text{monotone}(\text{Sgn}) \)

7. **correspondence of degree:**
   \( \forall \text{sgn} \in \text{Sgn} \). \( \text{degree}(\text{sgn}) = n \Rightarrow \text{sgn} \in \mu^n(\text{Sgn}) \land \text{sgn} \notin \mu^{n+1}(\text{Sgn}) \)

Property 1.1 states that \( \mu \) is injective; this implies the existence of a de-metacoding function \( \mu_{-1} \) (Property 1.2).

Property 1.3 reflects the fact that repeated metacoding creates a hierarchy of sets of metacoded components.

Property 1.4 states that programs metacoded at least \( n + 1 \) times (i.e., programs residing at level \( n + 1 \) or greater) can be adequately represented by values built using the metacode constructs of a program that has been metacoded \( n \) times (i.e., by a program residing at level \( n \)).

Property 1.5 states that \( \mu \) preserves syntactic categories (because it only increments indices).

Property 1.6 states that \( \mu \) preserves monotone components.

Property 1.7 formalizes the earlier intuitive description of degree — components of degree \( n \) correspond to objects that have been metacoded at most \( n \) times.
Trees:

\[ \mathcal{E} \vdash_{\text{ctr}} \text{true} \quad \Rightarrow (\text{true}, \mathcal{E}) \quad \mathcal{P}, \mathcal{E} \vdash_{\text{tree}} \text{true} \quad \Rightarrow \text{val} \]

\[ \mathcal{E} \vdash_{\text{ctr}} \text{false} \quad \Rightarrow (\text{false}, \mathcal{E}) \quad \mathcal{P}, \mathcal{E} \vdash_{\text{tree}} \text{true} \quad \Rightarrow \text{val} \]

\[ \mathcal{E} \vdash_{\text{exp}} \text{exp} \quad \Rightarrow \text{val} \quad \mathcal{P}, \mathcal{E}[\text{name} \mapsto \text{val}] \vdash_{\text{tree}} \text{tree} \quad \Rightarrow \text{val} \]

Constructions:

\[ \mathcal{E} \vdash_{\text{arg}} \text{arg} \quad \Rightarrow (\text{ATOM} \text{ atom}) \quad \mathcal{E} \vdash_{\text{arg}} \text{arg} \quad \Rightarrow (\text{ATOM} \text{ atom}) \]

\[ \mathcal{E} \vdash_{\text{arg}} \text{arg} \quad \Rightarrow (\text{ATOM} \text{ atom}) \quad \mathcal{E} \vdash_{\text{arg}} \text{arg} \quad \Rightarrow (\text{ATOM} \text{ atom}) \quad \text{atom}_1 \neq \text{atom}_2 \]

\[ \mathcal{E} \vdash_{\text{ctr}} \text{CONS} \text{ arg}_1 \text{ name}_1 \text{ name}_2 \quad \Rightarrow (\text{true}, \mathcal{E}[\text{name}_i \mapsto \text{val}_i]) \quad i = 1, 2 \]

\[ \mathcal{E} \vdash_{\text{ctr}} \text{CONS} \text{ arg}_1 \text{ name}_1 \text{ name}_2 \quad \Rightarrow (\text{false}, \mathcal{E}) \]

\[ \mathcal{E} \vdash_{\text{ctr}} \text{CONS} \text{ arg}_1 \text{ name}_1 \text{ name}_2 \quad \Rightarrow (\text{false}, \mathcal{E}) \]

\[ \mathcal{E} \vdash_{\text{ctr}} \text{CONS} \text{ arg}_1 \text{ name}_1 \text{ name}_2 \quad \Rightarrow (\text{false}, \mathcal{E}) \]

Fig. 8. Semantics of S-Graph-n (part 1)
Expressions:

\[
\begin{align*}
\mathcal{E} & \vdash^n_{exp} \exp_i^n \Rightarrow \text{val}_i^n \quad i = 1, 2 \\
\mathcal{E} & \vdash^n_{exp} (\text{CONS} \exp_i^n \exp_j^n) \Rightarrow (\text{CONS} \text{val}_i^n \text{val}_j^n)
\end{align*}
\]

\[
\begin{align*}
\mathcal{E} & \vdash^n_{exp} \exp_i^n \Rightarrow \text{val}_i^n \quad i = 1, 2, 3 \\
\mathcal{E} & \vdash^n_{exp} (\text{MC} \exp_i^n \exp_j^n \exp_k^n) \Rightarrow \text{val}_i^n \\
& \quad \text{(if val}_i^n = \text{pack}^n(\text{val}_i^n, \text{val}_j^n, \text{val}_k^n))
\end{align*}
\]

\[
\begin{align*}
\mathcal{E} & \vdash^n_{arg} \text{arg}^n \Rightarrow \text{val}^n \\
\mathcal{E} & \vdash^n_{mc} \text{mc}^n \Rightarrow \text{val}^n
\end{align*}
\]

Arguments:

\[
\begin{align*}
\mathcal{E} & \vdash^n_{arg} (\text{ATOM}^n \text{atom}) \Rightarrow (\text{ATOM}^n \text{atom}) \\
\mathcal{E} & \vdash^n_{arg} (\text{MV}^n \text{h name}) \Rightarrow (\text{MV}^n \text{h name})
\end{align*}
\]

\[
\begin{align*}
\mathcal{E} & \vdash^n_{arg} (\text{PV}^n \text{name}) \Rightarrow \mathcal{E}(\text{name})
\end{align*}
\]

Metocode (excerpts):

\[(m \geq 1 \text{ in the following rules})\]

\[
\begin{align*}
\mathcal{E} & \vdash^n_{mc} (\text{PV}^{n+m} \text{name}) \Rightarrow (\text{PV}^{n+m} \text{name})
\end{align*}
\]

\[
\begin{align*}
\mathcal{E} & \vdash^n_{mc} \exp_i^n \Rightarrow \text{val}_i^n \quad i = 1, 2, 3 \\
\mathcal{E} & \vdash^n_{mc} (\text{IF}^{n+m} \exp_i^n \exp_j^n \exp_k^n) \Rightarrow (\text{IF}^{n+m} \text{val}_i^n \text{val}_j^n \text{val}_k^n)
\end{align*}
\]

\[
\begin{align*}
\mathcal{E} & \vdash^n_{mc} \exp_i^n \Rightarrow \text{val}_i^n \quad i = 1, 2 \\
\mathcal{E} & \vdash^n_{mc} (\text{LET}^{n+m} \text{name} \exp_i^n \exp_j^n) \Rightarrow (\text{LET}^{n+m} \text{name} \text{val}_i^n \text{val}_j^n)
\end{align*}
\]

\[
\begin{align*}
\mathcal{E} & \vdash^n_{mc} (\text{MV}^{n+m} \exp_i^n \text{name}_1 \text{name}_2) \Rightarrow (\text{MV}^{n+m} \text{val}_i^n \text{name}_1 \text{name}_2)
\end{align*}
\]

Fig. 9. Semantics of S-Graph-n (part 2)

4.3 Semantics

Figures 8 and 9 present an operational semantics for S-Graph-n. Judgments for each syntactic category are parameterized by the reference level \(n\). For example, the derivability of the judgment

\[
\mathcal{P}, \mathcal{E} \vdash^n_{tree} \text{tree}^n \Rightarrow \text{val}^n
\]

signifies that given definitions \(\mathcal{P}\) and environment \(\mathcal{E}\), \(\text{tree}^n \in \text{Tree}[n]\) evaluates to \(\text{val}^n \in \text{Values}[n]\). The remaining judgments are similar. Definitions \(\mathcal{P}\) are not required in the remaining judgments since function calls can only occur in the syntactic category \(\text{Tree}[n]\). Evaluation of contractions \(\text{enr}^n\) returns \(\text{true}\) or \(\text{false}\) as well as a possibly updated environment.

The manipulation of metacode and metavariables is the most unique aspect of programming in S-Graph-n. We discuss this in detail below.

Metacode construction: Metacode components can be
\[
\begin{align*}
\text{unpack}^n((\text{IF}^m \ \text{sgn}_1 \ \text{sgn}_2)) &= \langle (\text{ATOM}^m \ p), (\text{ATOM}^m \ \text{IF}), \\
&\quad (\text{CONS}^m \ \text{sgn}_1 \ (\text{CONS}^m \ \text{sgn}_2 \ \text{sgn}_3)) \rangle \\
\text{unpack}^n((\text{LET}^m \ \text{name} \ \text{sgn}_1 \ \text{sgn}_2)) &= \langle (\text{ATOM}^m \ p), (\text{ATOM}^m \ \text{LET}), \\
&\quad (\text{CONS}^m \ (\text{ATOM}^m \ \text{name}) \ (\text{CONS}^m \ \text{sgn}_1 \ \text{sgn}_2)) \rangle \\
&\vdots \\
\text{unpack}^n((\text{CONS}^m \ \text{sgn}_1 \ \text{sgn}_2)) &= \langle (\text{ATOM}^m \ p), (\text{ATOM}^m \ \text{CONS}), (\text{CONS}^m \ \text{sgn}_1 \ \text{sgn}_2) \rangle \\
\text{unpack}^n((\text{MV}^m \ h \ \text{name})) &= \langle (\text{ATOM}^m \ p), (\text{ATOM}^m \ \text{MV}), \\
&\quad (\text{CONS}^m \ (\text{ATOM}^m \ h) \ (\text{ATOM}^m \ \text{name})) \rangle \\
\text{unpack}^n((\text{ATOM}^m \ \text{atom})) &= \langle (\text{ATOM}^m \ p), (\text{ATOM}^m \ \text{ATM}), (\text{ATOM}^m \ \text{atom}) \rangle \\
\end{align*}
\]

Note: The above definitions hold for all \( p, m, n \) such that \( p > 0 \) and \( m = n + p \)

Fig. 10. Packing and unpacking of metacode (excerpts)

(i) represented as literals,
(ii) created using the MC construct.

With (i), the index of the component (as well as elements of base syntax domains such as atom and name) is known statically, i.e., given as literal in the program. Evaluation of components (formalized by the Metacode rules of Figure 9) is similar to that of CONS. For example,\(^5\)

\[> \text{(eval 0 ' (let x (atom-1 foo) (if-1 x (atom bar) x)))} \]

\[\text{(if-1 (atom-1 foo) (atom bar) (atom-1 foo))} \]

This gives an effect similar to quasi-quotiation: a literal construct contains non-value components that may be further evaluated (this is the case with \text{x} in if-1 above).

With (ii), the index of the component (as well as base syntax domains such as atom and name) can be supplied dynamically, i.e., computed at runtime. MC takes three arguments: an atom indicating the index of the expression to be created, an atom indicating the tag of the component, and a tree of subcomponents. This is formalized by the rule for MC in Figure 9 and by the rules for pack in Figure 10. The value of the index atom must be greater than 0, and the number of subcomponents must correspond to the given tag — otherwise a runtime error occurs. The constraints on \( n, p, \) and \( m \) in Figure 10 indicate that the index manipulated in the MC and MC? constructs (the index corresponds to \( p \) in Figure 10) is relative to the reference level \( n \). Note that the reference level is 0 in the following example.

\[> \text{(eval 0 ' (let index (atom 1) (let tag (atom if) (let comp (cons (atom-1 foo)) (cons (atom bar) (atom-1 foo))))}) \]

\(^5\) The S-Graph-n system is written in Scheme, so data is supplied to the interpreter as S-expressions.
(mc index tag comp))))
(if-1 (atom-1 foo) (atom bar) (atom-1 foo))

One may question including both methods (i) and (ii) in the language. Method (i) is needed because metacoding must embed trees (active components) into values (passive components). Method (ii) is required to construct metacode where indices or tags are not known until run time (the usual situation in self-interpretation).

**Metacode destruction:** Metacode components are destructed using the `MC?` contraction.

```lisp
> (eval 0 '(let metacode (cons-1 (atom-1 a) (atom-1 b))
  (if (mc? metacode index tag comp)
  (cons index (cons tag comp))
  (atom false))))

(cons (atom 1) (cons (atom cons) (cons (atom-1 a) (atom-1 b)))
```

**Metavars:** As reflected in the definition of values (Figure 6), metavariables are canonical so their evaluation is trivial. Metavars are destructed using the `MV?` contraction.

```lisp
> (eval 0 '(if (mv? (mv 0 x) elev name)
  (cons elev name)
  (atom false)))

(cons (atom 0) (atom x))
```

`MV?` is the crucial predicate used in S-Graph-n self-applicable specializers since it determines whether a data object is known or unknown. Using conventional terminology, the test `(MV? arg name1 name2)` succeeds if `arg` represents a *dynamic* object. The use of `MV?` in self-application will be detailed in Section 5.

## 5 Metavars

High-level abstract presentations of metavars have been given elsewhere [7,27]. Here we describe the actual operations on metavars and illustrate the use of these operations in constructing biased generating extensions. Moreover, we give a semantic justification of these operations based on the formal semantics of S-Graph-n presented in Section 4.

### 5.1 Metavars attributes

A specializer written in S-Graph-n uses metavars as tags for representing unknown values in the program that it specializes. A metavar has three attributes which determine its semantics: *degree*, *domain*, and *elevation*.\(^6\)

\[
\begin{align*}
\text{degree}((\text{MV}\ n \ h \ name)) &= n \\
\text{domain}((\text{MV}\ n \ h \ name)) &= \text{Values}[n + h + 1] \\
\text{elevation}((\text{MV}\ n \ h \ name)) &= h
\end{align*}
\]

\(^6\) The reader should be warned that our definition of metavar degree differs slightly from other work [7,27]. In those works, our metavar of degree \(n\) has degree \(n + 1\).
Example program:

```lisp
(if (eq? x 'test-x)
  (if (eq? y 'test-y)
    'true-x-true-y
    'true-x-false-y)
  'false-x)
```

Initial call to the self-applicable specializer:

```lisp
(let tree (if-1 (eq?-1 (pv-1 x) (atom-1 test-x))
  (if-1 (eq?-1 (pv-1 y) (atom-1 test-y))
    (atom-1 true-x-true-y)
    (atom-1 true-x-false-y))
  (atom-1 false-x))
(let env (cons (cons (atom x) (cons (atom y) (atom nil))))
  (cons (atom test-x) (cons (mv 0 y) (atom nil))))
(let defs (cons (atom nil) (atom nil))
  (call (spec-start tree env defs))))
```

Result of specialization:

```lisp
(if-1 (eq?-1 (pv-1 y) (atom-1 test-y))
  (atom-1 true-x-true-y)
  (atom-1 true-x-false-y))
```

Fig. 11. Specialization example (analogous to the first Futamura projection)

*Degree* (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* restricts the domain of the metavariable to a particular set of values (this is motivated in detail below). Although both degree and elevation are numerical attributes, degree is an absolute characteristic, whereas elevation is a relative characteristic (it adjusts the domain relative to degree). Thus, elevation is unchanged by metacoding and de-metacoding (see Figure 7).

### 5.2 Metavariable examples

Intuitively, a metavariable of degree $n$ is used by a specializer $\text{spec}_n$ running at level $n$ to represent an unknown value in the program $\text{prog}_{n+1}$ which it is specializing. Specifically, an unknown input parameter $x$ for $\text{prog}_{n+1}$ will be bound to a metavariable in the symbolic environment used by $\text{spec}_n$ when specializing $\text{prog}_{n+1}$.

Consider the example program of Figure 11 where the free variables $x$ and $y$ represent input parameters. The self-applicable specializer is called with three arguments: a metacoded tree (e.g., the initial call for the program being specialized), an initial environment, and a list of metacoded definitions. Figure 11 presents the initial call used to specialize the example program where $x$ is bound to (atom test-x) and $y$ is unknown. The variable tree is bound to a metacoded version of the example program. The noteworthy point: the environment associates $x$ with (atom test-x) and $y$ with (mv 0 y) (i.e., a metavariable of degree 0 and elevation 0). Our example uses no functions, so the definition list defs is empty. The result of specialization (see Figure 11) is a metacoded tree.
(DEFINE (eval-val val env defs cont pe-cont)
  (IF (CUNES? cont cont-tag cont-rest)
    
    (IF (EQA? cont-tag 'eqa?1)
      (IF (MV? val elev name)
        (CALL (pe-cont val env defs cont pe-cont))
        (CALL (eval-eqa?-cont val env defs cont-rest pe-cont)))
      
      (IF (EQA? cont-tag 'eqa?r)
        (IF (MV? val elev name)
          (CALL (pe-cont val env defs cont pe-cont))
          (CALL (do-eqa? val env defs cont-rest pe-cont)))
      
      (IF (EQA? cont-tag 'cons?)
        (IF (MV? val elev name)
          (CALL (pe-cont val env defs cont pe-cont))
          (CALL (do-cons? val env defs cont-rest pe-cont)))))
    
  
  
Fig. 12. Specializer fragment that checks for unknowns in contractions

(which requires demetacoding before it can be executed).

Intuitively, spec uses MV? on x when interpreting the contraction (EQA? x 'test-x). Since x is bound to (atom test-x), MV? fails (signifying that x is known). spec uses MV? on y when interpreting the contraction (EQA? y 'test-y). Since y is bound to (mv 0 y), MV? succeeds (signifying that y is unknown).

Figure 12 presents a specializer code fragment that checks for unknowns in contractions. The specializer is written using first-order continuation-parsing. The displayed function eval-val dispatches on the continuation tag. For example, that tags eqa?1 and eqa?r indicate that val is the left and right component (respectively), of an eqa? contraction. In the example above, the false branch (CALL (eval-eqa?-cont ...)) associated with the eqa?1 continuation is executed when evaluating (EQA? x 'test-x); the true branch (CALL (pe-cont ...)) is executed when evaluating (EQA? y 'test-y). pe-cont performs appropriate residualization.

Second Futamura projection analogy: Now consider a situation analogous to the second Futamura projection where we attempt to produce a generating extension for the example program with inputs x and y.

<table>
<thead>
<tr>
<th>level 0</th>
<th>spec0</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 1</td>
<td>spec1</td>
</tr>
<tr>
<td>level 2</td>
<td>prog</td>
</tr>
</tbody>
</table>

There are now two copies of the self-applicable specializer: spec0 and spec1. spec0 is specializing spec1 which is specializing prog. spec1 and its associated initial call (see Figure 11) are metacoded once. Thus, the metavariable associated with y now has degree 1 and elevation 0. prog's variable x is no longer bound to data;
it now must be associated to a metavariable. Which metavariable? It should
not be a metavariable belonging to spec₁ i.e., a metavariable of degree 1 —
this would give x the same semantics as y, i.e., x would represent something
unknown to spec₁. This is not what we want if we desire to produce a generating
extension; we want x to appear as known to spec₁ but unknown to spec₀. Since
x is unknown to spec₀, it should be associated with metavariable belonging
to spec₀, i.e., a metavariable of degree 0.

Second Futamura projection analogy (failed attempt): Figure 13 pre-
sents the initial call to spec₀. test is now a metacoding of the initial call of
Figure 11. The only change is that the data (atom test-x) supplied for x has
been replaced by a program variable (pv-1 x). This adheres to our method of
representing input parameters as free variables; the data for x is now a parameter
to spec₁. env associates x with a metavariable (mv 0 x) following the conclusion
of the previous paragraph. def's is the lengthy list of metacoded definitions for
the self-applicable specializer.

Now consider what happens when x appears as the argument to the (EQA? x
'test-x) contraction in prog. spec₁ will use the MV? contraction (as shown under
the (EQA? cont-tag 'eqal)) alternative of Figure 12) to see whether the value
denoted by x is known or unknown. When spec₀ is interpreting this use of the
contraction MV? by spec₁, it recognizes one of its metavariables in the argument
of MV? and has no choice but to residualize it. Figure 13 shows the resulting
program (before post-processing and lifting). Space does not permit a detailed
explanation, but one can see the residualized MV? contraction and two branches
corresponding to the cases where x is known and unknown. This is an instance
of the "overly general compiler" problem [14, Chapter 7], and again is not what we
usually want in the second Futamura projection.

From a semantic point of view, spec₀ residualizes the contraction MV? as used
by spec₁ since the domain of its metavariable (MV? 0 x) is Values[0 + 0 + 1]. Note
that Values[1] (see Figure 6) includes metavariables of degree 1 (belonging to
spec₁) as well as other data of degree 1. So it is impossible for spec₀ to tell
whether or not a metavariable of degree 1 will flow into the argument of the
contraction MV? as used by spec₁.

Examining the domain of metavariable (MV? 0 x) also reveals a type mis-
mismatch: (MV? 0 x) ranges over Values[1] (i.e., over values computed at level 1),
but (MV? 0 x) is (indirectly) associated with the program variable x residing at
level 2! Thus, the domain of (MV? 0 x) is too general compared with the actual
set of values that may bind to x.

This mismatch can be corrected using the concept of elevation. In this case,
changing the elevation of (MV? 0 x) to 1, i.e., (MV? 1 x) gives the correct domain:
Values[0 + 1 + 1] = Values[2].

Second Futamura projection analogy (successful attempt): So say re-
place the metavariable (mv 0 x) with (mv 1 x) in the environment structure
ehv of Figure 13. Now consider what happens when x appears as the argument
to the contraction (EQA? x 'test-x) in prog. spec₁ will use the MV? contraction
(at level 1) as shown in Figure 12 to see whether the value denoted by x is known
or unknown. When spec₀ is interpreting this use of the contraction MV? by spec₁,
it recognizes one of its metavariables in the argument of MV?. However, spec₀ is
able to observe (using MV? at level 0) that (MV? 1 x) has domain Values[2]. Note
that Values[2] (see Figure 6) does not include metavariables of degree 1 (belong-
ing to \texttt{spec}_{c1}). So it is possible for \texttt{spec}_{c0} to tell that a metavariable of degree 1 will never flow into the argument of the contraction \texttt{MV} as used by \texttt{spec}_{c1}. Thus, \texttt{spec}_{c0} can conclude that the use of the contraction \texttt{MV} by \texttt{spec}_{c1} should fail, i.e., the “else” branch (\texttt{CALL (eval-equa? cont ...)} ) of the corresponding IF should be executed. Thus, from the point of view of \texttt{spec}_{c1}, (\texttt{FV x}) is not associated with a metavariable, but instead represents a known value. Figure 13 gives the program resulting from this appropriate use of elevation. This is the result we want: a “biased” generating extension.

5.3 Metavariable semantics

The example above clarifies the role of the \texttt{MV} contraction in detecting unknowns. Based on an understanding of metavariable domain, a specializer at level \( n \) properly interprets an \texttt{MV} contraction used at level \( n+1 \). We now define a predicate \( \phi_n \) which specifies the proper interpretation of an \texttt{MV} contraction performed at level \( n \). Intuitively, \( \phi_n(v) \equiv \text{“is } v \text{ unknown at level } n?\)”, or in other words, \( \phi_n(v) \equiv \text{“is } v \text{ a metavariable of degree } n?\)”.

\[
\begin{align*}
(1) & \quad \phi_n([\texttt{MV}^n h \texttt{name}]) = \text{true} \quad \text{for all } h \geq 0 \\
(2) & \quad \phi_n([\texttt{CONS} \texttt{val} \texttt{val}']) = \text{false} \\
(3) & \quad \phi_n([\texttt{ATOM} \texttt{atom}]) = \text{false} \\
(4) & \quad \phi_n(\texttt{v}) = \text{false} \quad \text{for all } v \in \texttt{Metacode}[n] \\
(5) & \quad \phi_n([\texttt{MV}^n' h \texttt{name}]) = \text{false} \quad \text{if } n' < n \text{ and } n \leq n' + h \\
(6) & \quad \phi_n([\texttt{MV}^n h \texttt{name}]) = \bot \quad \text{if } n' < n \text{ and } n > n' + h 
\end{align*}
\]

The first four cases are straightforward since the arguments to \( \phi_n \) are elements of \texttt{Values}[n] — the predicate is only true if a value at level \( n \) is a metavariable. Note that the case for \( \phi_n([\texttt{MV}^n h \texttt{name}]) \) where \( n' > n \) is covered by case (4) since such metavariables are metacode at level \( n \). From the perspective of level \( n' \), \( [\texttt{MV}^n h \texttt{name}]) \) is not a metavariable, but an encoded piece of program at level \( n' \).

The remaining cases are the interesting ones from the point of view of self-application.

Case (5) corresponds to situations where a metavariable appears to be a known value to the program running at level \( n \).

Justification: \texttt{domain}(\texttt{MV}^n h \texttt{name}]) = \texttt{Values}[p] \text{ for some } p > n, \text{ and } \texttt{Values}[p] \text{ does not include a metavariable of degree } n.

Case (6) corresponds to situations where it cannot be determined if the argument of \texttt{MV} is known or unknown at level \( n \).

Justification: \texttt{domain}(\texttt{MV}^n h \texttt{name}]) = \texttt{Values}[p] \text{ for some } p \leq n, \text{ and } \texttt{Values}[p] \text{ includes both metavariable of degree } n \text{ (unknowns) and known values. Intuitively, the \texttt{MV} contraction should be residualized in this case.}

6 Related Work

The ideas present in this paper have been heavily influenced by three concepts present in Turpin’s work [22,23]: metacoding, metavariables, and metasystem transition. Subsequently, these concepts have been formalized [7] and studied
in different contexts, e.g. [25]. The correct treatment of metacode was found essential in self-application [6] and this concept was singled out as elevation index [27].

The problem of self-application was the driving force behind the work on partial evaluation in the early eighties. The offline approach was originally introduced to avoid the generation of 'overly-general' compilers by self-application of a partial evaluator [15]; see also [14, Sec. 7.3]. Today off-line partial evaluation is the most developed approach to program specialization. This success lent itself to the hypothesis that self-application requires a binding-time analysis prior to specialization proper [16]. This hypothesis has been falsified where it was demonstrated that successful self-application without binding-time analysis is possible [6]. This insight has been used for the specialization of online partial evaluators [19]. A hybrid approach was used in [20] where off- and online partial evaluation was integrated; see also [4]. However, the power of off-line partial evaluation is limited by the approximations made during the binding-time analysis; e.g., off-line partial evaluation does not pass the KMP test. This led to the desire to self-apply stronger, online methods such as supercompilation [10,17] and partial deduction [5,11].

The idea of encoding expressions as data that can be manipulated as objects can be traced back to Gödel who used natural numbers for representing expressions in a first order language as data (to prove the well-known completeness and incompleteness theorems). Since then this methods has been used in logics and meta-mathematics to treat theories and proofs as formal objects and to prove properties about them. In computer science, especially in the area of logic programming, the encoding of programs has been studied under various names, e.g. naming relation [28,3]. Representing and reasoning about object level theories is an important field in logic and artificial intelligence (e.g. different encodings have been discussed in [12]) and has led to the development of logic languages that support declarative metaprogramming (e.g. the programming language Gödel [13]). A multilevel metalogic programming language has been suggested in [2], an approach similar to our multi-level metaprogramming environment, but directed towards the hierarchical organization of knowledge (e.g. for legal reasoning); it allows deductions on different metalevels. Level indexing was used in [8] to annotated operations in generating extensions with their binding-times which, together with the cogen approach, provided an efficient solution for multiple self-application of offline partial evaluation. In this paper we used level indexing of data to provide a basis for a multi-level metaprogramming environment which is independent of certain transformation paradigms. Recently, multi-level lambda-calculi were studied in [18].

7 Conclusion

We have attempted to clarify semantic and implementation concepts related to the use of metacoding and metavariables in metaprogramming. Several challenging problems lie ahead:

- implementation of metasystem jumps to increase speed of specialization,
- incorporation of stronger specialization techniques such as supercompilation into our self-applicable specializer,
• formalization of more powerful generalization techniques in our context (with metacode and metavariables), and
• development of a user environment for metaprogramming centered around MST scheme as a specification language.

We believe S-Graph-n is an appropriate vehicle for studying these foundational problems associated with hierarchies of online specialization systems. It seems well-suited as the basis of an simple experimental metaprogramming environment that embraces Turchin’s view of computation using metasystem transition.

Acknowledgments

Thanks to Sergei Abramov, Neil Jones, Andrei Klimov, Eric Ruf, Michael Sperber, and last but not least, Valentin Turchin for stimulating discussions on various topics of this paper. Special thanks to Kristian Nielsen for comments on an earlier version of this paper. Finally, we would like to thank the participants of the Dagstuhl Seminar on “Partial Evaluation” for various useful comments.

References


This article was processed using the L\textsc{\textcopyright}T\textsc{\textcopyright}X macro package with LLNCS style.
Initial call to the self-applicable specializer:

(let tree (let-1 tree
    (if-2 (eqa?-2 (pv-2 x) (atom-2 test-x))
        (if-2 (eqa?-2 (pv-2 y) (atom-2 test-y))
            (atom-2 true-x-true-y)
            (atom-2 true-x-false-y))
        (atom-2 false-x))
    (let-1 env (cons-1 (cons-1 (atom-1 x)
                        (atom-1 y))
        (cons-1 (pv-1 x)
            (atom-1 nil)))
    (let-1 defs (cons-1 (atom-1 nil) (atom-1 nil))
        (call-1 (spec-start (pv-1 tree)
            (pv-1 env)
            (pv-1 defs))))))

(let env (cons (cons (atom x) (atom nil)))
    (cons (mv 0 x) (atom nil))) ;; metavariable for x
(let defs ... ;; metacoded spec defs
    (call (spec-start tree env defs)))))))

Resulting program (before post-processing and lifting):

(if-1 (mv?-1 (mv-0 0 x) elev name)
    (if-2 (eqa?-2 (mv-0 0 x) (atom test-x)) ;; case: x unknown
        (if-2 (eqa?-2 (mv-1 0 y) (atom test-y))
            (atom true-x-true-y)
            (atom true-x-false-y))
        (atom false-x))
    (if-1 (eqa?-1 (mv-0 0 x) (atom test-x)) ;; case: x known
        (if-2 (eqa?-2 (mv-1 0 y) (atom test-y))
            (atom true-x-true-y)
            (atom true-x-false-y))
        (atom false-x)))))

Resulting program using appropriate elevation:

(if-1 (eqa?-1 (pv-1 x) (atom-1 test-x))
    (if-2 (eqa?-2 (pv-2 y) (atom-2 test-y))
        (atom-2 true-x-true-y)
        (atom-2 true-x-false-y))
    (atom-2 false-x))

Fig. 13. Specialization example (analogous to the second Futamura projection)