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 numerical indices which indicate the number of times that a program piece has been encoded.

Evaluation of S-Graph-n is formalized *via* an operational semantics. 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 metavariables 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 compilers and other program generators. A metasystem hierarchy is any situation where a program p_0 is manipulating (*e.g.*, interpreting, compiling, transforming) 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 supercompilation and partial deduction have not yet given satisfactory results for all Futamura projections (not to mention multiple self-application or the specializer projections). It seems harder to reason about the behavior of hierarchies of online systems for several reasons.

I. 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.

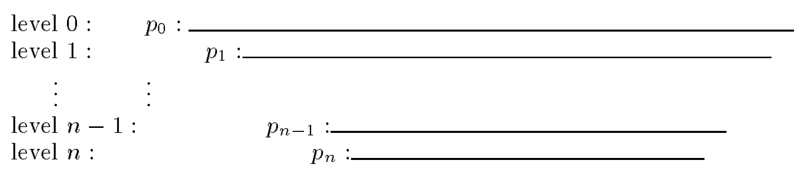

Fig. 1. MST Scheme of a metasystem hierarchy

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 a 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.

Programs:

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

Trees:

$tree ::=$ (`IF` $cntr\ tree\ tree$) | (`LET` $name\ exp\ tree$) | (`CALL` ($fname\ arg^*$)) | exp
$cntr ::=$ (`CONS?` $arg\ name\ name$) | (`EQA?` $arg\ arg$)
$exp ::=$ (`CONS` $exp\ exp$) | arg
$arg ::=$ (`ATOM` $atom$) | (`PV` $name$)

Fig. 2. Syntax of S-Graph

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?` arg_1 arg_2) — tests the equality of two atoms denoted by arg_1 and arg_2. If the arguments are non-atomic then the test is undefined.
- (`CONS?` $exp\ h\ t$) — if the value of *exp* is a pair (`CONS` val_1 val_2), then the test succeeds and the variable h is bound to val_1 and the variable t to val_2; 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 (CONS? x head tail)
       (LET headbag (CONS 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 metavariables may be metacoded many times. For example, p_n 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 p_n 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 corner stone of Turchin's approach: the construction of hierarchies of arbitrary height is taken as the basis for program analysis and transformation [21] (in contrast to logics and mathematics which usually deal with two-level hierarchies). For example, the well-known Futamura projections make use of a three-level hierarchy of metasystems (*i.e.*, program specializers). To make multi-level hierarchies practical, one needs facilities for satisfying the following requirements.

1. *Efficient encoding: low space consumption and minimal overhead for manipulating 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 transformation. 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 Gödel [13] provides built-in metaprogramming facilities for two-level hierarchies.

$$\begin{aligned}
\mu\{(\mathtt{IF}\ cntr\ tree_1\ tree_2)\} &= (\mathtt{CONS}\ (\mathtt{ATOM\ IF})\ (\mathtt{CONS}\ \mu\{cntr\}\ (\mathtt{CONS}\ \mu\{tree_1\}\ \mu\{tree_2\}))) \\
\mu\{(\mathtt{LET}\ name\ exp\ tree)\} &= (\mathtt{CONS}\ (\mathtt{ATOM\ LET})\ (\mathtt{CONS}\ (\mathtt{ATOM}\ name)\ (\mathtt{CONS}\ \mu\{exp\}\ \mu\{tree\}))) \\
&\dots \\
\mu\{(\mathtt{CONS}\ exp_1\ exp_2)\} &= (\mathtt{CONS}\ (\mathtt{ATOM\ CONS})\ (\mathtt{CONS}\ \mu\{exp_1\}\ \mu\{exp_2\})) \\
\mu\{(\mathtt{ATOM}\ atom)\} &= (\mathtt{CONS}\ (\mathtt{ATOM\ ATOM})\ (\mathtt{ATOM}\ atom))
\end{aligned}$$

Fig. 4. S-Graph metacoding (excerpts)

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

$$\overline{\text{(CONS(ATOMa)(ATOMb))}} = \text{(CONS (ATOM CONS) (CONS (CONS (ATOM ATOM) (ATOM a)) (CONS (ATOM ATOM) (ATOM b))))}$$

and metacoded twice is

$$\overline{\overline{\text{(CONS(ATOMa)(ATOMb))}}} = \text{(CONS (ATOM CONS) (CONS (CONS (ATOM ATOM) (ATOM CONS)) (CONS (ATOM CONS) (CONS (CONS (ATOM CONS) (CONS (CONS (ATOM ATOM) (ATOM ATOM)) (CONS (ATOM ATOM) (ATOM a)))) (CONS (ATOM CONS) (CONS (CONS (ATOM ATOM) (ATOM ATOM)) (CONS (ATOM ATOM) (ATOM b))))))))}$$

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 subcomponent (*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.

Compositional. 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 `(CONS (ATOM a) (ATOM b))` two times.

$$\overline{\overline{\text{(CONS(ATOMa)(ATOMb))}}} = \text{''(CONS (ATOM a) (ATOM b))}$$

Suppose we wish to replace the subexpression `(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 `list`) to rebuild the enclosing data structure.

$$\overline{\overline{\text{(CONS}}}\ \text{x}\ \overline{\overline{\text{(ATOMb)}}} = \text{(LIST ''CONS x ''(ATOM b))}$$

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 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 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

$$\overline{\text{(CONS(ATOMa)(ATOMb))}} = \text{(CONS-1 (ATOM-1 a) (ATOM-1 b))}$$

and metacoded twice is

$\overline{\overline{\texttt{(CONS(ATOMa)(ATOMb))}}}$ = (CONS-2 (ATOM-2 a) (ATOM-2 b))

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 hierarchy) 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.

$$(\texttt{IF}^2\ (\texttt{CONS}^1\ (\texttt{ATOM}^3\ \texttt{foo})\ (\texttt{ATOM}^1\ \texttt{bar}))\ (\texttt{PV}^3\ x)\ (\texttt{PV}^4\ y)) \qquad (1)$$

$$(\texttt{IF}^2\ (\texttt{EQA?}^2\ (\texttt{ATOM}^2\ \texttt{foo})\ (\texttt{ATOM}^2\ \texttt{bar}))\ (\texttt{PV}^3\ x)\ (\texttt{PV}^4\ y)) \qquad (2)$$

$$(\texttt{IF}^3\ (\texttt{EQA?}^2\ (\texttt{ATOM}^3\ \texttt{foo})\ (\texttt{ATOM}^3\ \texttt{bar}))\ (\texttt{CONS}^1\ (\texttt{ATOM}^2\ \texttt{1})\ (\texttt{ATOM}^2\ \texttt{1}))\ (\texttt{PV}^4\ y)) \qquad (3)$$

Intuitively, the S-Graph-n components are indexed building blocks for constructing 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 situation in metaprogramming applications: well-formedness must be enforced by the programmer.[2]

The *index* of an S-Graph-n component *sgn* is the level number attached to the outermost construct of *sgn*. 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 *sgn* (denoted *degree*(*sgn*)) is the smallest index occurring in *sgn*. 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:

$$
\begin{array}{rcl}
sgn & \in & Sgn \\
sgn & ::= & (\mathtt{IF}^n\ sgn\ sgn\ sgn) \mid (\mathtt{LET}^n\ name\ sgn\ sgn) \mid \ldots \mid \\
 & & (\mathtt{CONS?}^n\ sgn\ name\ name) \mid (\mathtt{EQA?}^n\ sgn\ sgn) \mid \\
 & & (\mathtt{MC?}^n\ sgn\ name\ name\ name) \mid (\mathtt{MV?}^n\ sgn\ name\ name) \mid \ldots \mid \\
 & & (\mathtt{CONS}^n\ sgn\ sgn) \mid (\mathtt{ATOM}^n\ atom) \mid (\mathtt{PV}^n\ name) \mid \\
 & & (\mathtt{MC}^n\ sgn\ sgn\ sgn) \mid (\mathtt{MV}^n\ h\ name) \qquad \textit{for any } n, h \geq 0
\end{array}
$$

Trees at level n:

$$
\begin{array}{rcl}
tree^n & \in & Tree[n] \\
tree^n & ::= & (\mathtt{IF}^n\ cntr^n\ tree^n\ tree^n) \mid (\mathtt{LET}^n\ name\ exp^n\ tree^n) \mid \\
 & & (\mathtt{CALL}^n\ (fname\ arg^*)) \mid exp^n
\end{array}
$$

$$
\begin{array}{rcl}
cntr^n & \in & Cntr[n] \\
cntr^n & ::= & (\mathtt{CONS?}^n\ arg^n\ name\ name) \mid (\mathtt{EQA?}^n\ arg^n\ arg^n) \mid \\
 & & (\mathtt{MC?}^n\ arg^n\ name\ name\ name) \mid (\mathtt{MV?}^n\ arg^n\ name\ name)
\end{array}
$$

$$
\begin{array}{rcl}
exp^n & \in & Exp[n] \\
exp^n & ::= & (\mathtt{CONS}^n\ exp^n\ exp^n) \mid (\mathtt{MC}^n\ arg^n\ arg^n\ arg^n) \mid arg^n \mid mc^n
\end{array}
$$

$$
\begin{array}{rcl}
arg^n & \in & Arg[n] \\
arg^n & ::= & (\mathtt{ATOM}^n\ atom) \mid (\mathtt{PV}^n\ name) \mid (\mathtt{MV}^n\ h\ name) \qquad \textit{for any } h \geq 0
\end{array}
$$

Metacode at level n:

$$
\begin{array}{rcl}
mc^n & \in & Metacode[n] \\
mc^n & ::= & (\mathtt{IF}^{n+m}\ exp^n\ exp^n\ exp^n) \mid (\mathtt{LET}^{n+m}\ name\ exp^n\ exp^n) \mid \ldots \mid \\
 & & (\mathtt{CONS}^{n+m}\ exp^n\ exp^n) \mid (\mathtt{ATOM}^{n+m}\ atom) \mid (\mathtt{PV}^{n+m}\ name) \mid \\
 & & (\mathtt{MC}^{n+m}\ exp^n\ exp^n\ exp^n) \mid (\mathtt{MV}^{n+m}\ h\ name) \qquad \textit{for any } m \geq 1, h \geq 0
\end{array}
$$

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 *sgn* is *monotone* if the indices of subcomponents are the same or are increasing as one descends down the abstract syntax tree of *sgn*. Formally, all leaf components are monotone; a non-leaf component *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 *sgn*. In the components above, (2) is monotone; (1) and (3) are not. Given a set $S \subseteq Sgn$, $monotone(S)$ denotes the subset of monotone components of S.

Elements of $Tree[n] \subset Sgn$ 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 $Metacode[n] \subset Tree[n]$ 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 $Metacode[0]$. However, only (2) is an element of $Metacode[1]$ since $(\mathtt{ATOM}^1\ \mathtt{bar})$ in (1) and

$$
\begin{array}{rcl}
\mathit{val}^n & \in & \mathit{Values}[n] \\
\mathit{val}^n & ::= & (\mathtt{ATOM}^n\ \mathit{atom}) \mid (\mathtt{MV}^n\ h\ \mathit{name}) \mid (\mathtt{CONS}^n\ \mathit{val}^n\ \mathit{val}^n) \mid \\
 & & (\mathtt{IF}^{n+m}\ \mathit{val}^n\ \mathit{val}^n\ \mathit{val}^n) \mid (\mathtt{LET}^{n+m}\ \mathit{name}\ \mathit{val}^n\ \mathit{val}^n) \mid \ldots \mid \\
 & & (\mathtt{MC}^{n+m}\ \mathit{val}^n\ \mathit{val}^n\ \mathit{val}^n) \mid (\mathtt{MV}^{n+m}\ h\ \mathit{name}) \qquad \textit{for any } h \geq 0,\ m \geq 1
\end{array}
$$

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

($\mathtt{CONS}^1$ ($\mathtt{ATOM}^2$ `1`) ($\mathtt{ATOM}^2$ `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 ($\mathtt{MC}^n$ arg^n arg^n arg^n), ($\mathtt{MC?}^n$ arg^n *name name name*) are added to construct and destruct metacode. Metavariables ($\mathtt{MV}^n$ h *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 $\mathit{Values}[n] \subset \mathit{Exp}[n]$ (Figure 6). Intuitively, values at level n are either atoms, metavariables, 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 *reify* 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 μ 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.

$$\begin{aligned}
\mu\{(\mathtt{IF}^n\ sgn_1\ sgn_2\ sgn_3)\} &= (\mathtt{IF}^{n+1}\ \mu\{sgn_1\}\ \mu\{sgn_2\}\ \mu\{sgn_3\}) \\
\mu\{(\mathtt{LET}^n\ name\ sgn_1\ sgn_2)\} &= (\mathtt{LET}^{n+1}\ name\ \mu\{sgn_1\}\ \mu\{sgn_2\}) \\
&\dots \\
\mu\{(\mathtt{CONS}^n\ sgn_1\ sgn_2)\} &= (\mathtt{CONS}^{n+1}\ \mu\{sgn_1\}\ \mu\{sgn_2\}) \\
\mu\{(\mathtt{ATOM}^n\ atom)\} &= (\mathtt{ATOM}^{n+1}\ atom) \\
\mu\{(\mathtt{MV}^n\ h\ name)\} &= (\mathtt{MV}^{n+1}\ h\ name)
\end{aligned}$$

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

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

Property 1 (Properties of μ)

1. **injectivity:** $\forall sgn_1, sgn_2 \in Sgn\,.\ \ sgn_1 \neq sgn_2 \ \Rightarrow\ \mu\{sgn_1\} \neq \mu\{sgn_2\}$

2. **left inverse μ_{-1}:** $\forall sgn \in Sgn.\ \ sgn \ = \ \mu_{-1}\{\mu\{sgn\}\}$

3. **embedding:** $\forall n \geq 0\,.\ \ \mu^n\{Sgn\} \supset \mu^{n+1}\{Sgn\}$

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

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

6. **preservation of monotonicity:** $\mu\{monotone(Sgn)\} \subset monotone(Sgn)$

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

Property 1.1 states that μ is injective; this implies the existence of a demetacoding function μ_{-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 μ preserves syntactic categories (because it only increments indices).

Property 1.6 states that μ 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:

$$\frac{\mathcal{E} \vdash^n_{cntr} \mathit{cntr}^n \Longrightarrow \langle \mathbf{true} , \mathcal{E}' \rangle \qquad \mathcal{P} , \mathcal{E}' \vdash^n_{tree} \mathit{tree}^n_1 \Longrightarrow \mathit{val}^n}{\mathcal{P} , \mathcal{E} \vdash^n_{tree} (\texttt{IF}^n\ \mathit{cntr}^n\ \mathit{tree}^n_1\ \mathit{tree}^n_2) \Longrightarrow \mathit{val}^n}$$

$$\frac{\mathcal{E} \vdash^n_{cntr} \mathit{cntr}^n \Longrightarrow \langle \mathbf{false} , \mathcal{E}' \rangle \qquad \mathcal{P} , \mathcal{E}' \vdash^n_{tree} \mathit{tree}^n_2 \Longrightarrow \mathit{val}^n}{\mathcal{P} , \mathcal{E} \vdash^n_{tree} (\texttt{IF}^n\ \mathit{cntr}^n\ \mathit{tree}^n_1\ \mathit{tree}^n_2) \Longrightarrow \mathit{val}^n}$$

$$\frac{\mathcal{E} \vdash^n_{exp} \mathit{exp}^n \Longrightarrow \mathit{val}^n_1 \qquad \mathcal{P} , \mathcal{E}[\mathit{name} \mapsto \mathit{val}^n_1] \vdash^n_{tree} \mathit{tree}^n \Longrightarrow \mathit{val}^n}{\mathcal{P} , \mathcal{E} \vdash^n_{tree} (\texttt{LET}^n\ \mathit{name}\ \mathit{exp}^n\ \mathit{tree}^n) \Longrightarrow \mathit{val}^n}$$

$$\frac{\begin{array}{c} \mathcal{E} \vdash^n_{arg} \mathit{arg}^n_i \Longrightarrow \mathit{val}^n_i \\ \mathcal{P}(\mathit{fname}) = \langle (\mathit{name}_1, ..., \mathit{name}_m) , \mathit{tree}^n \rangle \\ \mathcal{P} , [\mathit{name}_i \mapsto \mathit{val}^n_i] \vdash^n_{tree} \mathit{tree}^n \Longrightarrow \mathit{val}^n \qquad \text{i=1,..,m} \end{array}}{\mathcal{P} , \mathcal{E} \vdash^n_{tree} (\texttt{CALL}^n\ (\mathit{fname}\ \mathit{arg}^n_1, ...\ \mathit{arg}^n_m)) \Longrightarrow \mathit{val}^n}$$

Contractions:

$$\frac{\mathcal{E} \vdash^n_{arg} \mathit{arg}^n_1 \Longrightarrow (\texttt{ATOM}^n\ \mathit{atom}) \qquad \mathcal{E} \vdash^n_{arg} \mathit{arg}^n_2 \Longrightarrow (\texttt{ATOM}^n\ \mathit{atom})}{\mathcal{E} \vdash^n_{cntr} (\texttt{EQA?}^n\ \mathit{arg}^n_1\ \mathit{arg}^n_2) \Longrightarrow \langle \mathbf{true} , \mathcal{E} \rangle}$$

$$\frac{\mathcal{E} \vdash^n_{arg} \mathit{arg}^n_1 \Longrightarrow (\texttt{ATOM}^n\ \mathit{atom}_1) \qquad \mathcal{E} \vdash^n_{arg} \mathit{arg}^n_2 \Longrightarrow (\texttt{ATOM}^n\ \mathit{atom}_2) \qquad \mathit{atom}_1 \neq \mathit{atom}_2}{\mathcal{E} \vdash^n_{cntr} (\texttt{EQA?}^n\ \mathit{arg}^n_1\ \mathit{arg}^n_2) \Longrightarrow \langle \mathbf{false} , \mathcal{E} \rangle}$$

$$\frac{\mathcal{E} \vdash^n_{arg} \mathit{arg}^n \Longrightarrow (\texttt{CONS}^n\ \mathit{val}^n_1\ \mathit{val}^n_2)}{\mathcal{E} \vdash^n_{cntr} (\texttt{CONS?}^n\ \mathit{arg}^n\ \mathit{name}_1\ \mathit{name}_2) \Longrightarrow \langle \mathbf{true} , \mathcal{E}[\mathit{name}_i \mapsto \mathit{val}^n_i] \rangle \qquad i = 1, 2}$$

$$\frac{\mathcal{E} \vdash^n_{arg} \mathit{arg}^n \Longrightarrow \mathit{val}^n \qquad \mathit{val}^n \neq (\texttt{CONS}^n\ \mathit{val}^n_1\ \mathit{val}^n_2)}{\mathcal{E} \vdash^n_{cntr} (\texttt{CONS?}^n\ \mathit{arg}^n\ \mathit{name}_1\ \mathit{name}_2) \Longrightarrow \langle \mathbf{false} , \mathcal{E} \rangle}$$

$$\frac{\mathcal{E} \vdash^n_{arg} \mathit{arg}^n \Longrightarrow \mathit{val}^n \qquad \mathbf{unpack}^n(\mathit{val}^n) = \langle \mathit{val}^n_1, \mathit{val}^n_2, \mathit{val}^n_3 \rangle}{\mathcal{E} \vdash^n_{cntr} (\texttt{MC?}^n\ \mathit{arg}^n\ \mathit{name}_1\ \mathit{name}_2\ \mathit{name}_3) \Longrightarrow \langle \mathbf{true} , \mathcal{E}[\mathit{name}_i \mapsto \mathit{val}^n_i] \rangle \qquad i = 1, 2, 3}$$

$$\frac{\mathcal{E} \vdash^n_{arg} \mathit{arg}^n \Longrightarrow \mathit{val}^n \qquad \mathbf{unpack}^n(\mathit{val}^n)\ \mathit{undefined}}{\mathcal{E} \vdash^n_{cntr} (\texttt{MC?}^n\ \mathit{arg}^n\ \mathit{name}_1\ \mathit{name}_2\ \mathit{name}_3) \Longrightarrow \langle \mathbf{false} , \mathcal{E} \rangle}$$

$$\frac{\begin{array}{c} \mathcal{E} \vdash^n_{arg} \mathit{arg}^n \Longrightarrow (\texttt{MV}^n\ h\ \mathit{name}) \\ \mathcal{E}' = \mathcal{E}[\mathit{name}_1 \mapsto (\texttt{ATOM}^n\ h), \mathit{name}_2 \mapsto (\texttt{ATOM}^n\ \mathit{name})] \end{array}}{\mathcal{E} \vdash^n_{cntr} (\texttt{MV?}^n\ \mathit{arg}^n\ \mathit{name}_1\ \mathit{name}_2) \Longrightarrow \langle \mathbf{true} , \mathcal{E}' \rangle}$$

$$\frac{\mathcal{E} \vdash^n_{arg} \mathit{arg}^n \Longrightarrow \mathit{val}^n \qquad \mathit{val}^n \neq (\texttt{MV}^n\ h\ \mathit{name})}{\mathcal{E} \vdash^n_{cntr} (\texttt{MV?}^n\ \mathit{arg}^n\ \mathit{name}_1\ \mathit{name}_2) \Longrightarrow \langle \mathbf{false} , \mathcal{E} \rangle}$$

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

Expressions:

$$\frac{\mathcal{E} \vdash^n_{exp} exp^n_i \Longrightarrow val^n_i \quad i = 1,2}{\mathcal{E} \vdash^n_{exp} (\texttt{CONS}^n\ exp^n_1\ exp^n_2) \Longrightarrow (\texttt{CONS}^n\ val^n_1\ val^n_2)}$$

$$\frac{\mathcal{E} \vdash^n_{exp} exp^n_i \Longrightarrow val^n_i \quad i = 1,2,3}{\mathcal{E} \vdash^n_{exp} (\texttt{MC}^n\ exp^n_1\ exp^n_2\ exp^n_3) \Longrightarrow val^n} \quad (\text{if } val^n = \mathbf{pack}^n(val^n_1, val^n_2, val^n_3))$$

$$\frac{\mathcal{E} \vdash^n_{arg} arg^n \Longrightarrow val^n}{\mathcal{E} \vdash^n_{exp} arg^n \Longrightarrow val^n} \qquad \frac{\mathcal{E} \vdash^n_{mc} mc^n \Longrightarrow val^n}{\mathcal{E} \vdash^n_{exp} mc^n \Longrightarrow val^n}$$

Arguments:

$$\mathcal{E} \vdash^n_{arg} (\texttt{ATOM}^n\ atom) \Longrightarrow (\texttt{ATOM}^n\ atom) \qquad \mathcal{E} \vdash^n_{arg} (\texttt{MV}^n\ h\ name) \Longrightarrow (\texttt{MV}^n\ h\ name)$$

$$\mathcal{E} \vdash^n_{arg} (\texttt{PV}^n\ name) \Longrightarrow \mathcal{E}(name)$$

Metacode (excerpts): ($m \geq 1$ in the following rules)

$$\mathcal{E} \vdash^n_{mc} (\texttt{PV}^{n+m}\ name) \Longrightarrow (\texttt{PV}^{n+m}\ name)$$

$$\frac{\mathcal{E} \vdash^n_{mc} exp^n_i \Longrightarrow val^n_i \quad i = 1,2,3}{\mathcal{E} \vdash^n_{mc} (\texttt{IF}^{n+m}\ exp^n_1\ exp^n_2\ exp^n_3) \Longrightarrow (\texttt{IF}^{n+m}\ val^n_1\ val^n_2\ val^n_3)}$$

$$\frac{\mathcal{E} \vdash^n_{mc} exp^n_i \Longrightarrow val^n_i \quad i = 1,2}{\mathcal{E} \vdash^n_{mc} (\texttt{LET}^{n+m}\ name\ exp^n_1\ exp^n_2) \Longrightarrow (\texttt{LET}^{n+m}\ name\ val^n_1\ val^n_2)}$$

$$\frac{\mathcal{E} \vdash^n_{mc} exp^n \Longrightarrow val^n}{\mathcal{E} \vdash^n_{mc} (\texttt{MV?}^{n+m}\ exp^n\ name_1\ name_2) \Longrightarrow (\texttt{MV?}^{n+m}\ val^n\ name_1\ name_2)}$$

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

$$\mathcal{P}, \mathcal{E} \vdash^n_{tree} tree^n \Longrightarrow val^n$$

signifies that given definitions $\mathcal{P}$ and environment $\mathcal{E}$, $tree^n \in Tree[n]$ evaluates to $val^n \in Values[n]$. 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 $Tree[n]$. Evaluation of contractions $cntr^n$ returns **true** or **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

$$
\begin{array}{rcl}
\mathbf{unpack}^n((\mathtt{IF}^m\ sgn_1\ sgn_2\ sgn_3)) & = & \langle(\mathtt{ATOM}^n\ p),\ (\mathtt{ATOM}^n\ \mathtt{IF}),\\
 & & (\mathtt{CONS}^n\ sgn_1\ (\mathtt{CONS}^n\ sgn_2\ sgn_3))\rangle\\
\mathbf{unpack}^n((\mathtt{LET}^m\ name\ sgn_1\ sgn_2)) & = & \langle(\mathtt{ATOM}^n\ p),\ (\mathtt{ATOM}^n\ \mathtt{LET}),\\
 & & (\mathtt{CONS}^n\ (\mathtt{ATOM}^n\ name)\ (\mathtt{CONS}^n\ sgn_1\ sgn_2))\rangle\\
 & \dots & \\
\mathbf{unpack}^n((\mathtt{CONS}^m\ sgn_1\ sgn_2)) & = & \langle(\mathtt{ATOM}^n\ p),\ (\mathtt{ATOM}^n\ \mathtt{CONS}),\ (\mathtt{CONS}^n\ sgn_1\ sgn_2)\rangle\\
\mathbf{unpack}^n((\mathtt{MV}^m\ h\ name)) & = & \langle(\mathtt{ATOM}^n\ p),\ (\mathtt{ATOM}^n\ \mathtt{MV}),\\
 & & (\mathtt{CONS}^n\ (\mathtt{ATOM}^n\ h)\ (\mathtt{ATOM}^n\ name))\rangle\\
\mathbf{unpack}^n((\mathtt{ATOM}^m\ atom)) & = & \langle(\mathtt{ATOM}^n\ p),\ (\mathtt{ATOM}^n\ \mathtt{ATOM}),\ (\mathtt{ATOM}^n\ atom)\rangle
\end{array}
$$

$$
\begin{array}{c}
\mathbf{pack}^n\langle(\mathtt{ATOM}^n\ p),\ (\mathtt{ATOM}^n\ \mathtt{IF}),\ (\mathtt{CONS}^n\ sgn_1\ (\mathtt{CONS}^n\ sgn_2\ sgn_3))\rangle = (\mathtt{IF}^m\ sgn_1\ sgn_2\ sgn_3)\\
\dots\\
\mathbf{pack}^n\langle(\mathtt{ATOM}^n\ p),\ (\mathtt{ATOM}^n\ \mathtt{ATOM}),\ (\mathtt{ATOM}^n\ atom)\rangle = (\mathtt{ATOM}^m\ atom)
\end{array}
$$

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]

```
> (eval 0 '(let x (atom-1 foo) (if-1 x (atom bar) x)))

(if-1 (atom-1 foo) (atom bar) (atom-1 foo))
```

This gives an effect similar to quasi-quotation: a literal construct contains non-value components that may be further evaluated (this is the case with `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.

```
> (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 is 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.

```
   > (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))))
```

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

```
   > (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* $name_1$ $name_2$) succeeds if *arg* represents a *dynamic* object. The use of `MV?` in self-application will be detailed in Section 5.

5 Metavariables

High-level abstract presentations of metavariables have been given elsewhere [7,27]. Here we describe the actual operations on metavariables 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 Metavariable attributes

A specializer written in S-Graph-n uses metavariables as tags for representing unknown values in the program that it specializes. A metavariable has three attributes which determine its semantics: *degree*, *domain*, and *elevation*.[6]

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

[6] The reader should be warned that our definition of metavariable degree differs slightly from other work [7,27]. In those works, our metavariable of degree n has degree $n+1$.

Example program:

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

Initial call to the self-applicable specializer:

```
(let tree (if-1 (eqa?-1 (pv-1 x) (atom-1 test-x))
                (if-1 (eqa?-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:

```
(if-1 (eqa?-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 demetacoding (see Figure 7).

5.2 Metavariable examples

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

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 (CONS? cont cont-tag cont-rest)
         :
      (IF (EQA? cont-tag 'eqa?l)
          (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-passing. The displayed function `eval-val` dispatches on the continuation tag. For example, that tags `eqa?l` 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?l` 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`.

level 0: $spec_0$: ______________________________
level 1: $spec_1$: ________________________
level 2: *prog* : ____________

There are now two copies of the self-applicable specializer: $spec_0$ and $spec_1$. $spec_0$ is specializing $spec_1$ which is specializing *prog*. $spec_1$ 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 $spec_1$ (*i.e.*, a metavariable of degree 1) — this would give `x` the same semantics as `y`, *i.e.*, `x` would represent something unknown to $spec_1$. This is not what we want if we desire to produce a generating extension; we want `x` to appear as known to $spec_1$ but unknown to $spec_0$. Since `x` is unknown to $spec_0$, it should be associated with metavariable belonging to $spec_0$, *i.e.*, a metavariable of degree 0.

Second Futamura projection analogy *(failed attempt)*: Figure 13 presents the initial call to $spec_0$. `tree` 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 adhears to our method of representing input parameters as free variables; the data for `x` is now a parameter to $spec_1$. `env` associates `x` with a metavariable `(mv 0 x)` following the conclusion of the previous paragraph. `defs` 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*. $spec_1$ will use the `MV`? contraction (as shown under the `(EQA? cont-tag 'eqa?1)` alternative of Figure 12) to see whether the value denoted by `x` is known or unknown. When $spec_0$ is interpreting this use of the contraction `MV`? by $spec_1$, 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 an 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, $spec_0$ residualizes the contraction `MV`? as used by $spec_1$ since the domain of its metavariable (`MV`^{0} 0 `x`) is *Values*$[0+0+1]$. Note that *Values*[1] (see Figure 6) includes metavariables of degree 1 (belonging to $spec_1$) as *well as other data* of degree 1. So it is impossible for $spec_0$ to tell whether or not a metavariable of degree 1 will flow into the argument of the contraction `MV`? as used by $spec_1$.

Examining the domain of metavariable (`MV`^{0} 0 `x`) also reveals a *type mismatch*: (`MV`^{0} 0 `x`) ranges over *Values*[1] (*i.e.*, over values computed at level 1), but (`MV`^{0} 0 `x`) is (indirectly) associated with the program variable `x` residing at level 2! Thus, the domain of (`MV`^{0} 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} 0 `x`) to 1, *i.e.*, (`MV`^{0} 1 `x`) gives the correct domain: *Values*$[0+1+1]$ = *Values*[2].

Second Futamura projection analogy *(successful attempt)*: So say replace the metavariable `(mv 0 x)` with `(mv 1 x)` in the environment structure `env` of Figure 13. Now consider what happens when `x` appears as the argument to the contraction `(EQA? x 'test-x)` in *prog*. $spec_1$ 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 $spec_0$ is interpreting this use of the contraction `MV`? by $spec_1$, it recognizes one of its metavariables in the argument of `MV`?. *However*, $spec_0$ is able to observe (using `MV`? at level 0) that (`MV`^{0} 1 `x`) has domain *Values*[2]. Note that *Values*[2] (see Figure 6) *does not* include metavariables of degree 1 (belonging to $spec_1$). So it *is possible* for $spec_0$ to tell that a metavariable of degree 1 will never flow into the argument of the contraction `MV?` as used by $spec_1$. Thus, $spec_0$ can conclude that the use of the contraction `MV?` by $spec_1$ should fail, *i.e.*, the "else" branch `(CALL (eval-eqa?-cont ...))` of the corresponding `IF` should be executed. Thus, from the point of view of $spec_1$, `(PV 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 `MV?` contraction in detecting unknowns. Based on an understanding of metavariable *domain*, a specializer at level n properly interprets an `MV?` contraction used at level $n+1$. We now define a predicate ϕ_n which specifies the proper interpretation of an `MV?` contraction performed at level n. Intuitively, $\phi_n(v) \equiv$ "is v unknown at level n?", or in other words, $\phi_n(v) \equiv$ "is v a metavariable of degree n?'.

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

The first four cases are straightforward since the arguments to ϕ_n are elements of *Values*$[n]$ — the predicate is only true if a value at level n is a metavariable. Note that the case for $\phi_n((\texttt{MV}^{n'}\ h\ name))$ where $n' > n$ is covered by case (4) since such metavariables are metacode at level n. From the perspective of level n', $(\texttt{MV}^{n'}\ h\ name)$ 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: $domain((\texttt{MV}^{n'}\ h\ name)) = Values[p]$ for some $p > n$, and $Values[p]$ does not include a metavariable of degree n.

Case (6) corresponds to situations where it cannot be determined if the argument of `MV?` is known or unknown at level n.
Justification: $domain((\texttt{MV}^{n'}\ h\ name)) = Values[p]$ for some $p \leq n$, and $Values[p]$ includes both metavariable of degree n (unknowns) and known values. Intuitively, the `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 Turchin'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 offline 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

1. S.M. Abramov. *Metavychislenija i ikh prilozhenija (Metacomputation and its applications)*. Nauka, Moscow, 1995. (In Russian).
2. Jonas Barklund. A basis for a multilevel metalogic programming language. Technical Report 81, Uppsala University, Dept. of Computing Science, 1994.
3. Jonas Barklund. Metaprogramming in logic. Technical Report 80, Uppsala University, Dept. of Computing Science, 1994.
4. Charles Consel and Siau Cheng Khoo. Parameterized partial evaluation. *ACM TOPLAS*, 15(3):463–493, 1993.
5. Hiroshi Fujita and Koichi Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2-3):91–118, June 1988.
6. Robert Glück. Towards multiple self-application. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 309–320, New Haven, Connecticut, 1991. ACM Press.
7. Robert 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.
8. Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In S.D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.
9. Robert Glück and Andrei V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filè, and G. Rauzy, editors, *Static Analysis. Proceedings. Lecture Notes in Computer Science, Vol. 724*, pages 112–123. Springer-Verlag, 1993.
10. Robert Glück and Valentin F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the ISSAC '90 (Tokyo, Japan)*, pages 286–287. ACM Press, 1990.
11. Corin A. Gurr. *A self-applicable partial evaluator for the logic programming language Gödel*. Ph.d. thesis, University of Bristol, 1994.

12. Patricia Hill and John Gallagher. Meta-programming in logic programming. Technical Report 94.22, School of Computer Studies, University of Leeds, 1994.
13. Patricia Hill and John W. Lloyd. *The Gödel Programming Language.* MIT Press, Cambridge, Massachusetts, 1994.
14. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
15. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. Lecture Notes in Computer Science 202*, pages 124–140. Springer-Verlag, 1985.
16. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
17. Andrei Nemytykh, Victoria Pinchuck, and Valentin F. Turchin. A self-applicable supercompiler. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation. Proceedings*, Lecture Notes in Computer Science, page to appear. Springer-Verlag, 1996.
18. Flemming Nielson and Hanne R. Nielson. Multi-level lambda-calculi: an algebraic description. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation. Proceedings*, Lecture Notes in Computer Science, page to appear. Springer-Verlag, 1996.
19. Eric Ruf and Daniel Weise. On the specialization of online program specializers. *Journal of Functional Programming*, 3(3):251–281, 1993.
20. Michael Sperber. Self-applicable online partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation. Proceedings*, Lecture Notes in Computer Science, page to appear. Springer-Verlag, 1996.
21. Valentin F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, 1979.
22. Valentin F. Turchin. The language Refal, the theory of compilation and metasystem analysis. Courant Computer Science Report 20, Courant Institute of Mathematical Sciences, New York University, Feb. 1980 1980.
23. Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
24. Valentin F. Turchin. *Refal-5, Programming Guide and Reference Manual.* New England Publishing Co., Holyoke, Massachusetts, 1989.
25. Valentin F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.
26. V.F. Turchin. Programmirovanie na jazyke Refal: I. Neformal'noe vvedenie v programmirovanie na jazyke Refal. (Programming in the language Refal: I. Informal introduction to programming in the language Refal). Preprint 41, Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow, 1971. (In Russian).
27. V.F. Turchin and A.P. Nemytykh. Metavariables: their implementation and use in program transformation. Technical Report CSc. TR 95-012, City College of the City University of New York, 1995.
28. Frank van Harmelen. Definable naming relations in meta-level systems. In A. Pettorossi, editor, *Meta-Programming in Logic. Proceedings*, volume 649 of *Lecture Notes in Computer Science*, pages 89–104, Uppsala, Sweden, 1992. Springer-Verlag.

This article was processed using the LaTeX 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)
                                        (cons-1 (atom-1 y)
                                                (atom-1 nil)))
                                (cons-1 (pv-1 x)
                                        (cons-1 (mv-1 0 y)
                                                (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)