Internally Typed Second-Order Term Graphs

Wolfram Kahl

Institut für Softwaretechnologie, Fakultät für Informatik
Universität der Bundeswehr München, D-85577 Neubiberg, Germany
`kahl@informatik.unibw-muenchen.de`

Abstract. We present a typing concept for second-order term graphs that does not consider the types as an external add-on, but as an integral part of the term graph structure. This allows a homogeneous treatment of term-graph representations of many kinds of typing systems, including second-order λ-calculi and systems of dependent types. Applications can be found in interactive systems and as typed intermediate representation for example in compilers.

1 Introduction

Term graphs have originally been introduced as efficient representations of terms. The key to this efficiency is the possibility of *sharing* of what on the term side are equal substructures. Linear notation systems for term graphs (such as in some functional programming languages) often use names bound by e.g. `where`-clauses to express sharing; this kind of binding is perceived as different from that introduced by λ-abstractions as represented in term graphs e.g. by Wadsworth, the inventor of graph reduction [12], the difference being that the names bound by `where`-clauses do not appear in the graph, but those bound by λ-abstractions do.

In previous work [3] we took the step to consider *both* uses of bound variable names only as *coding* of structure that can be made *explicit* with an appropriate definition of term graphs. The structure element encoded by `where`-bindings is traditionally explicit in term graphs as the possibility that nodes have several predecessors; the structure element encoded by the names and scopes of λ-bound variables is in the first instance that of *variable binding*, a function that assigns every bound variable its binder, and in the second instance that of *variable identity*, an equivalence relation among variables which makes explicit which variable occurrences belong to the same variable.

This principle of rigourously making structure explicit is now applied to typing in this paper.

In conventional typing systems, types are assigned to *terms.* But when formally reasoning about terms, usually the corresponding *abstract syntax trees* are considered instead. Now trees can not only be considered as a free algebra, where trees are built from constituents (subtrees) via operators, they can also be considered as a special kind of directed graphs, where a node may have successor nodes. Therefore, the types of terms in conventional systems correspond to types of nodes in term graphs, and where conventional systems relate the types of terms with the types of their immediate constituents, a term graph typing system should relate the type of a node with the type of its successor nodes.

Furthermore, types usually are again terms of some language, so when we represent terms and their subterms as nodes in a term graph, we can equally represent types as nodes in a term graph of types. Now there are systems that use (sublanguages of) the same language for programs and types, and that even allow references from programs to types (as in second-order λ-calculi) or from types to program terms (as in dependent types), so it seems only natural to consider a program and its type as parts of *one* term graph that is enriched with an additional *typing function* from program nodes to their type nodes.

After establishing some notation in Sect. 2, we define internally typed second-order term graphs in Sect. 3 and their homomorphisms in Sect. 4. The principles of an appropriate framework for typing systems are laid out in Sect. 5, with some details about the typing of multi-node variables left to Sect. 6. Section 7 shows how our typed term graphs can be considered to result from a kind of algebraic graph grammars. Finally we shortly present a few more complicated typing systems in Sect. 8 for giving an impression of the power of our formalism.

2 Notation

In our formalisation, we frequently use relational operations since this allows very clear and concise formalisations in the context of graphs, see e.g. [9, 2].

For many purposes we use parts of the Z-notation [10], most notably for set comprehensions where Z uses the pattern "$\{$*signature* $|$ *predicate* $\bullet$ *term*$\}$" instead of the otherwise frequently observed "$\{$*term* $|$ *predicate*$\}$". So we have as an example $\{n : \mathbb{N} \mid n < 4 \bullet n^2\} = \{0, 1, 4, 9\}$. If the predicate is constantly true, then we can also write "$\{$*signature* $\bullet$ *term*$\}$", e.g., $\{x : \mathbb{B} \bullet (x, x)\} = \{(\mathsf{True}, \mathsf{True}), (\mathsf{False}, \mathsf{False})\}$; if the *term* is just the tuple of the variables introduced in the *signature*, then another possibility is "$\{$*signature* $|$ *predicate*$\}$", e.g. $\{x, y : \mathbb{B} \mid x \neq y\} = \{(\mathsf{True}, \mathsf{False}), (\mathsf{False}, \mathsf{True})\}$. Quantification uses the same patterns; here most frequently the predicate is omitted, so we have for example $\forall x : \mathbb{N} \bullet x + 1 > x$. The powerset of a set A is written $\mathbb{P}.A$.

The set of relations between two sets A and B is written $A \leftrightarrow B$ and is equal to the power set of the cartesian product: $(A \leftrightarrow B) := \mathbb{P}.(A \times B)$. The set of univalent relations or partial functions from A to B is written $A \nrightarrow B$, and that of total functions or mappings is written $A \rightarrow B$. Application of a function $f : A \nrightarrow B$ to an argument $x : A$ is written "$f\,.x$" and is only used if the argument is known to be in the domain of the function: $x \in \mathsf{dom}.f$, where for any relation R the domain of R is $\mathsf{dom}.R := \{(x, y) : R \bullet x\}$, and the range of R is $\mathsf{ran}.R := \{(x, y) : R \bullet y\}$.

The set A^* is the set of finite sequences of elements of A; these sequences are considered to be partial functions of type $\mathbb{N} \nrightarrow A$ with contiguous domain which, when nonempty, always includes zero; therefore, if l is a sequence, then $l.i$ denotes the $(i + 1)$-th element of l. For any set A, the function $\mathsf{len} : A^* \rightarrow \mathbb{N}$ calculates the length of sequences.

The identity relation on a set A is $\mathbb{I}_A : A \rightarrow A$, and we usually just write $\mathbb{I}$. For two sets A and B, the universal relation is $\mathbb{T}_{A,B} := A \times B$ and the empty relation is $\perp\!\!\!\perp_{A,B} := \emptyset$; again we usually just write $\mathbb{T}$ and $\perp\!\!\!\perp$.

For two relations $R, S : A \leftrightarrow B$, their intersection is $R \cap S$ and their union is $R \cup S$; inclusion is written $R \subseteq S$. The complement of R is $\overline{R}$. The converse of R is the relation $R^{\smile} : B \leftrightarrow A$, defined by $R^{\smile} := \{(x, y) : R \bullet (y, x)\}$.

For two relations $R : A \leftrightarrow B$ and $S : B \leftrightarrow C$, their composition is $R{;}S : A \leftrightarrow C$ with $R{;}S := \{(x, y) : R; (u, z) : S \mid y = u \bullet (x, z)\}$. The transitive closure of a homogeneous relation $R : A \leftrightarrow A$ is R^+, and the reflexive transitive closure is R^*.

When a relation $R : A \leftrightarrow A$ is considered as a **graph**, a node $y : A$ is *reachable* from another node $x : A$ if and only if $(x, y) \in R^*$. The relation R is *acyclic* if $R^+ \subseteq \overline{I}$. A node r is a *source* if $r \notin \mathsf{ran}.R$, and r is a *root* if it is the only source (at least in the DAG setting).

3 Term Graph Definition

In comparison with the untyped graphs of [3,5], we present a simplified formalisation; for the sake of brevity we do not fully formalise obvious concepts. However, we immediately present a definition for *typed* term graphs. For this purpose, we first formalise our view of typing without reference to any concrete typing system, just regarding the typing function as another term graph component.

The "lexical material" which we fill our term graph structure with is essentially the same as for second-order terms (or metaterms, introduced by Klop [7]), but we do not introduce a separate class of binders, and what usually is called "function symbol" is called "constant constructor" here, since we want to stress the contrast with variables:

Definition 3.1. A **term graph alphabet** is a tuple $(\mathcal{L}, A, \mathcal{C}, \mathcal{B}, \mathcal{M})$ with the set $\mathcal{L}$ of *node labels*, the *arity function* $A : \mathcal{L} \to \mathbb{N}$, and a partition of $\mathcal{L}$ into the sets $\mathcal{C}$ of labels for *constant constructors*, $\mathcal{B}$ for *bindable variables*, and $\mathcal{M}$ for *metavariables.* $\square$

In the following we assume a fixed term graph alphabet $(\mathcal{L}, A, \mathcal{C}, \mathcal{B}, \mathcal{M})$.

The main differences between the following definition and those of [3,5] — besides the introduction of typing — are that we here only consider finite acyclic graphs and that the set of edge labels is fixed as the set of natural numbers.

Definition 3.2. A **term graph** is a tuple $G = (\mathcal{N}, L, S, D, B, W, T)$ with

- $\mathcal{N}$, the finite **node set**,
- $L : \mathcal{N} \to \mathcal{L}$, the **node labelling** function,
- $S : \mathcal{N} \to \mathcal{N}^*$, the **successor** function with $L{;}A = S{;}\mathsf{len}$, i.e., the length of the successor list of each node has to be the arity of its label,
- $D : \mathcal{N} \leftrightarrow \mathcal{N}$, the **associated relation**, $D := \{(x, l) : S; y : \mathsf{ran}.l \bullet (x, y)\}$; obviously D is not a primitive component but derived from S; it is listed here for its importance,
- $T : \mathcal{N} \nrightarrow \mathcal{N}$, the partial **typing** function, where $(D \cup T)$ has to be acyclic,
- $B : \mathcal{N} \nrightarrow \mathcal{N}$, the **binding** function, where for $(x, b) \in B$, the *bound variable* x has to have a label in $\mathcal{B}$, the *binder* b a label in $\mathcal{C}$, and b *dominates*[1] x in the graph induced by $(D \cup T)$,

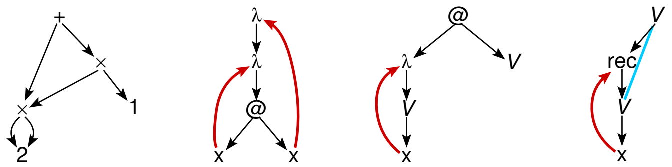
[1] In graph theory, a node b dominates another node x, if for every node a and every path from a to x either b lies on that path or a is reachable from b. Domination therefore implies reachability.

– $W : \mathcal{N} \leftrightarrow \mathcal{N}$, the **variable identity**, a *partial equivalence relation*[1] defined exactly on *variables*, i.e. on nodes with labels from $\mathcal{B} \cup \mathcal{M}$. The variable identity has to be compatible with the labelling: $W;L \subseteq L$, and with the binding[2]: $W;B \subseteq B$.

Roots are considered wrt. $(D \cup T)$, and the **type part** of a typed term graph is the set $\mathsf{ran}.(T;(D \cup T)^*)$ containing all nodes reachable from typing nodes. □

At first sight it might seem strange that we did not impose any restriction on the interplay of the typing function with the other term graph components, most notably with variable identity. But as we shall see in Sects. 6 and 8, different typing systems open up very different possibilities and also impose different restrictions, so that it does not make sense to impose restrictions on the level of the term graph definition, especially since we still lack the machinery to formulate most of the useful restrictions.

Terms corresponding to a rooted term graph are easily recovered by unfolding recursively along the D-Paths from the root — creation of a unique name for every variable is the easiest means to ensure preservation of the binding and variable identity structure. Consider the following examples (of untyped graphs, i.e., of graphs with empty typing), where successor edges are black arrows with their sequence indicated by the left-to-right order of their attachment to their source node; binding edges are drawn in red resp. as thick, dark grey, usually curved arrows, and an irreflexive kernel of variable identity is indicated by blue resp. thick medium gray lines:



The first two correspond to the terms "$2 \cdot 2 + 2 \cdot 2 \cdot 1$" and "$\lambda x.\lambda f. f\, x$" from arithmetic resp. λ-calculus. For the last two, let us assume that A and B are metavariables — in HOPS (see [6]), where the pictures have been produced, arity is part of the node label, so that unary and zero-ary metavariables all are drawn with the label V, but according to their arity they should be considered as different labels $V_0, V_1 : \mathcal{M}$ in the examples. The last two term graphs then correspond to the metaterms "$(\lambda\, x. B[x])\, A$" and "$B[\mathsf{rec}\ x. B[x]]$", respectively.

The concept of free variables is easy to transfer to term graphs; especially important is the relation between a binder and those nodes below which its bound variable occurs freely:

Definition 3.3. A variable node x is **free below** a node a, if there is a $(D \cup T)$-path from a to x such that no binder of x lies on that path; if in this constellation x is bound by b, then b **encapsulates** a. The **encapsulation** $C : \mathcal{N} \leftrightarrow \mathcal{N}$ relates b with a exactly when b encapsulates a. □

---

[1] A partial equivalence relation is a symmetric and transitive relation.

[2] This condition, $W;B \subseteq B$, means that if any node in an equivalence class wrt. W is bound by some binder, then all nodes in that class are bound by the same binder — note the conciseness and elegance of the relational formulation!

4 Homomorphy

In conventional term graph formalisms there are no bound variables, and variables corresponding to our metavariables are always zero-ary. Therefore, when considering homomorphisms based on node mappings, the image of such a variable is the whole subgraph starting at the image node of the variable node. In the case of second-order term graphs however, there are metavariables with successors, and their images have to *stop* before the image nodes of their successors. Therefore we introduce:

Definition 4.1. An **interval** in a graph G is a pair $(t,b) : (\mathcal{N} \times (\mathbb{N} \nrightarrow \mathcal{N}))$ consisting of a *top node t* and a finite *lower border b*, which should be considered as a partial node sequence. The **inner nodes** of the interval (t,b) are those nodes that are $(D \cup T)$-reachable from t via paths on which there lies no node of $\mathsf{ran}.b$. The interval (t,b) is **coherent** if all nodes in the lower border (i.e. all nodes in $\mathsf{ran}.b$) are $(D \cup T)$-reachable from t. □

Not every interval is a reasonable candidate for being image of metavariables; conditions corresponding to "no capture of variables" have to be fulfilled. Auxiliary concepts for dealing with this issue are:

Definition 4.2. An interval is **consistent**, if all nodes encapsulated by inner nodes are inner nodes. The **encapsulation skeleton** of an interval is the set of those inner nodes, from which a node in the lower border can be reached via a $(B \cup (D \cup T))$-path. □

The most important use of term graph homomorphisms is to serve as matchings from rule sides into application graphs for transformation or rewriting. In the term context, matching is usually defined as "there exist a context and a (second-order) substitution, such that the result of inserting the substituted rule side into the context is α-equivalent to the application term" — with the definitions of substitution application and insertion into contexts taking care of avoiding "variable capture". In term graphs, a more direct approach is necessary, and in second-order term graphs the structure that is not there is almost as important as the structure that is there, so the conditions for structure preservation take on an unusual shape, and for avoiding "variable capture" several special conditions are needed. While in [3] we worked only with total functions and in [5] we went all the way to possibly partial and multivalent relations, here we just present a definition using potentially partial functions. This still allows a reasonably simple treatment of the images of metavariables:

Definition 4.3. If for two graphs G_1 and G_2, a partial function $F : \mathcal{N}_1 \nrightarrow \mathcal{N}_2$ is given, then the **image interval** for a metavariable node $m : \mathcal{N}_1$ with $m \in \mathsf{dom}.F$ and $L_1.m \in \mathcal{M}$ is defined to be the interval $(F.m, (S_1.m);F)$. □

The fact that we only consider acyclic graphs and homomorphisms with rooted domain graphs helps considerably to keep the conditions simple:

Definition 4.4. A **metavariable base** in a term graph G is a set v of metavariable nodes that is closed under variable identity. □

Definition 4.5. A v**-fitting** from a term graph G_1 with metavariable base v to a term graph G_2 is a function $F : \mathcal{N}_1 \nrightarrow \mathcal{N}_2$ that (we let $F_0 := F \cap \overline{v \times \mathcal{N}_2}$ be the *constant part* of F relative to v)

– *preserves labels:* $F_0^{\smile};L_1 \subseteq L_2$,
– *preserves successors:* $\forall(n1, n2) : F_0 \bullet S_1.n_1 \subseteq (S_2.n_2);F^{\smile}$,
– is *coherent* and *consistent:* the image interval of every metavariable node in $\mathsf{dom}.F$ is coherent and consistent,
– *strictly preserves binding:* $F_0^{\smile};B_1 = B_2;F_0^{\smile}$,
– *strictly preserves variables:* $W_1;F_0 = F_0;W_2$,
– *respects equally bound variables:* $F_0;W_2;F_0^{\smile} \cap B_1;B_1^{\smile} \subseteq W_1$,
– *respects free variables:* $F_0;W_2;F_0^{\smile} \cap \overline{B_1;\mathbb{T}} \subseteq W_1$,
– *controls variables:* if for some metavariable node $m : \mathcal{N}_1$ in $\mathsf{dom}.F$, a variable node x_2 in the image interval of m is free below $F.m$, then there is no (variable) node $x_1 : \mathcal{N}_1$ in $\mathsf{dom}.F_0$, such that $(F_0.x_1, x_2) \in W_2$.
– *preserves typing:* $F^{\smile};T_1 \subseteq T_2;F^{\smile}$, and *preserves typelessness:* $F;T_2 \subseteq T_1;F$. □

Note that the conditions for the typing must not be restricted to the constant part of F.

For **linear** term graphs, i.e. where the variable identity restricted to metavariables is trivial, this is already the definition of homomorphisms; for non-linear term graphs we need a concept of "isomorphism up to sharing" between image intervals of metavariables, so we define:

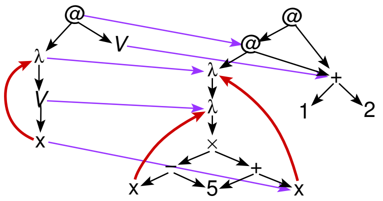
Definition 4.6. A **correspondence** between two intervals (t_1, b_1) and (t_2, b_2) in a graph G is a relation $H : \mathcal{N} \leftrightarrow \mathcal{N}$ on the node set fulfilling the following conditions:
– H exactly covers the inner nodes: $\mathsf{dom}.H$ is the set of the inner nodes of (t_1, b_1), and $\mathsf{ran}.H$ is the set of the inner nodes of (t_2, b_2),
– H preserves upper borders: if H is non-empty, it relates t_1 exactly to t_2 and vice versa,
– H preserves node labels: $H;L \subseteq L$,
– H preserves successors: $(H \parallel I);S \subseteq S;(H \cup b_1^{\smile};b_2)$,
– H preserves bindings on internally bound nodes:
$$H;B \cap \mathbb{T};H \subseteq B;H \subseteq H;B \quad \text{and} \quad H^{\smile};B \cap \mathbb{T};H^{\smile} \subseteq B;H^{\smile} \subseteq H^{\smile};B,$$
– H respects the distinctness of nodes internally bound by the same constructor:
$$H^{\smile};W;H \cap \mathbb{T};H;B \cap B;B^{\smile} \subseteq W,$$
and so does $H^{\smile}$,
– H preserves variables: $H;W \cap \mathbb{T};H = W;H \cap H;\mathbb{T}$,
– H stays within the same variable for variables that are not internally bound:
$$H \cap \overline{B;H;\mathbb{T}} \cap W;\mathbb{T} \subseteq W,$$
– H preserves typing: $H;T \cap \mathbb{T};H = T;H \cap H;\mathbb{T}$. □

It is relatively easy to see that for a correspondence H, its converse $H^{\smile}$ is a correspondence, too. Also the identity relation restricted to the inner nodes of a consistent interval obviously is a correspondence, and it can be proved that the composition of two correspondences is again a correspondence. Existence of correspondences between consistent image intervals of metavariables is therefore an equivalence relation, and it is natural to define:

Definition 4.7. A v**-homomorphism** is a v-fitting where for every two metavariable nodes in $W_1 \cap (v \times v)$ there is a correspondence between their image intervals. □

Without further restrictions, these homomorphisms are not composable, but in [3] we have shown that this is not necessary for being able to define a sound rewriting concept (see also [4]).

As an example homomorphism (indicated by the thin, dark grey (violet) arrows) we show an untyped β-redex to the right.



5 Well-Typed Term Graphs

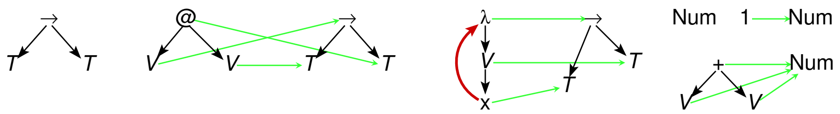
We now introduce a means to distinguish well-typed term graphs. The system we propose is a system that could equally well be employed for untyped term graphs; there it would allow to make distinctions that are not covered by Def. 3.2, such as which node labels are allowed as binders, and below which successors bound variables may occur (a step in the direction of the general "binding structures" of [11]). As shown here, the system still may be used towards these purposes, although its main motivation is to ascertain legal typing.

We define "typing elements" as schema graphs that encode what "locally legal" should mean for a graph:

Definition 5.1. A **typing element** is a typed term graph G which either is rooted or has all its sources related to each other by the variable identity, and where all successors of the source nodes are metavariables and all successors of those metavariables are bound by the root node. Such a typing element is said to be **for** the label of its root node. □

Here we assume that all bindable variables have zero arity; otherwise the same conditions would have to be enforced for them as for the sources.

We provide three example typing elements for simply-typed λ-calculus and three more for arithmetics — the typing function is denoted by green resp. thin, light grey arrows:



Obviously typing elements closely correspond to typing rules in type derivation systems as they can be found e.g. in the *pure type systems* of [1]. For reasons of space we refrain from formalising this relation, since then we would have to introduce the formalism of pure type systems, too; we only state:

Proposition 5.1. As long as there is no reduction among types, there is a one-to-one correspondence between typing elements and typing rules of pure type systems. □

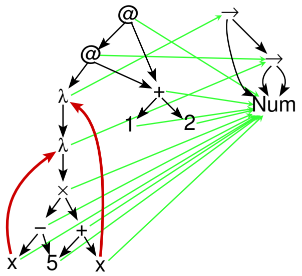
Just as only a set of rules defines a language in pure type systems, only a set of typing elements defines a typed term graph language:

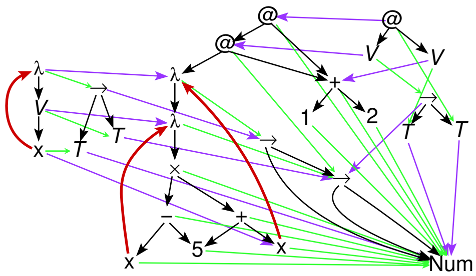
Definition 5.2. A **term graph language** is a set $\mathcal{T}$ of typing elements, such that $\mathcal{T}$ contains at most one typing element for each node label $l : \mathcal{C}$. □

The restriction that there is at most one typing element for every node label might be relaxed for implementing a kind of overloading, but we shall not pursue that possibility in this paper. In contrast to most typing systems, that build on some kind of derivation or inference, we use a more simultaneous concept here; we simply define "globally legal" as "locally legal everywhere":

Definition 5.3. A typed term graph G is **well-typed** wrt. a term graph language $\mathcal{T}$ if for every node $n : \mathcal{N}$ of G there is a typing element τ for $L.n$ and a homomorphism from τ into G that maps a source node of τ to n. □

Note that a typing element is necessary for *every* node, and not only for every typed node — this exactly corresponds to the situation in pure type systems, where a trivial type " □ " is given to those terms that correspond to term graph nodes without type. Otherwise this definition would for example not be able to ensure consistency in the presence of dependent types. For graphically conveying the idea behind our definition of well-typedness, we draw a typed version of the β-redex example from below Def. 4.7 once separately and once together with two example homomorphisms from the typing elements for function application and λ-abstraction.





For being able to discuss properties of term graph languages, we first introduce a few classifications of typing elements:

Definition 5.4. A typing element is called
- **separated**, iff it contains no $(D \cup B)$-edges between typed and untyped nodes,
- **well-typed wrt.** $\mathcal{T}$, iff its type part is well-typed wrt. $\mathcal{T}$.
- **first-order**, iff all metavariables contained in the type part are zero-ary. □

For the kind of typing elements we have seen so far, a principal type property is easy to establish:

Theorem 5.1 (Principal Types). If the term graph language $\mathcal{T}$ fulfils the following conditions:
- all typing elements are separated and first-order,
- the relation $Q : \mathcal{L} \leftrightarrow \mathcal{L}$ with $Q := \{G : \mathcal{T}; l : \mathsf{ran}.(T_G;D_G^*;L_G) \bullet (L_G.r_G,\ l)\}$, where r_G denotes the root of graph G, is acyclic,
- if G is a typing element for l in $\mathcal{T}$, then G is well-typed wrt. the sub-language of $\mathcal{T}$ for the labels that are reachable from l in Q,

then there is a **principal type** for every graph in the following sense: Let the program part of a graph be the untyped subgraph induced by all typed nodes (closing it wrt. successors, binding, and bound variables); then among all well-typed graphs with isomorphic program parts there is always one from which there is a homomorphism to every other. □

Term graph languages with these restrictions correspond to simple parametric polymorphism; full Hindley-Milner polymorphism with `let`-polymorphism requires features of type abstraction, see Sect. 8.2. Other principal type results can also be carried over to the term graph setting.

6 Typing Elements for Variables

The definition 5.3 of well-typedness implies that there also have to be typing elements for variables. Since variables may consist of many nodes, and since the condition of strict variable preservation (Def. 4.5) demands at least as many variable nodes in the source as in the target, the term graph language (Def. 5.2) which is used as the basis for well-typedness may severely restrict possible variable occurrences.

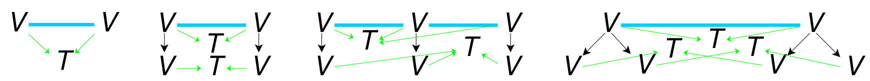
For example, if for the bound variable label x there is only one typing element $x \longrightarrow T$, this implies that on x-nodes the variable identity is trivial. HOPS currently restricts bound variables in this way, which is only natural in a term DAG setting.

With metavariables, on the other hand, such a restriction is not feasible anymore, at least not for metavariables with successors. For untyped (i.e., type-level) metavariables without successors, HOPS also provides only one typing element, since in HOPS the type part of every term DAG is kept maximally identified.

Obviously, if more than one node per variable is going to be allowed, it makes little sense to impose other restrictions on the number of those nodes, so we need infinitely many typing elements for every metavariable label concerned.

For untyped metavariables with untyped successors, this is not a big problem since their typing elements will have empty typing and the only restriction one might impose is that the different nodes of one variable shared their successors. However, such a restriction seems to be difficult to motivate.

For the other cases there are more possibilities. Let us restrict our attention here to typed metavariables with typed successors, which is also the only kind currently fully supported by HOPS. In HOPS, for every positive integer n and every natural number i a typing element is assumed for n i-ary typed metavariable nodes all related by the variable identity; the successors are all distinct zero-ary typed metavariables with trivial variable identity, and there are $i + 1$ zero-ary untyped metavariables with trivial variable identity, one for the sources and one for every successor index. We show the typing elements for $(n, i) \in \{(2, 0), (2, 1), (3, 1), (2, 2)\}$:



Well-typedness with such a term graph language implies very simple restrictions on the typing function which can also be expressed directly: All nodes of a variable have to have the same type: $W;T \subseteq T$, and the corresponding successors of the different nodes of one variable have the same type: $\forall (x, y) : W \bullet (S.x);T = (S.y);T$.

We shall see more general typing elements for metavariables in Sect. 8 together with the frameworks that make them necessary.

7 Construction Step Homomorphisms — Towards Typed Term Graph Grammars

In this section we show how stepwise term graph construction — as for example in an interactive system — can be viewed as a sequence of homomorphisms. These "construction step homomorphisms" belong to a few simple classes, which together can be considered to define a special kind of graph grammar which is closely related to algebraic graph grammars, since it relies on category-theoretic concepts.

It is important to note that in this section homomorphisms are restricted to homomorphisms between well-typed graphs wrt. some term graph language.

The simplest kind of homomorphism that is sometimes needed during construction does not instantiate any metavariables:

Definition 7.1. An **identification** is a homomorphism with empty metavariable base; i.e., it is an $\emptyset$-homomorphism. □

Identifications stand in a one-to-one correspondence with congruence relations on term graphs; for every identification F, the equivalence relation $(F;F^{\smile})^{*}$ is a congruence, and for every congruence, the quotient projection is an identification.

Since we restrict ourselves to finite graphs, every identification can be broken down into a sequence of "primitive identifications", the correspondences of which are the correspondence closure of two-node sets.

Isomorphisms between typed term graphs may instantiate metavariables in that the only visible change that can be brought about by isomorphisms is consistent permutation of the successors of metavariables.

Similar to the identification of two nodes, there is the possibility of unification of two nodes. But since this in general brings about the non-determinism of second-order unification, we have to restrict ourselves to simple deterministic cases:

Definition 7.2. A **replacement** of a metavariable node v with another node r which is not a successor of v is a homomorphism F which unifies v and r and which uniquely factorises any other such homomorphism. □

It is comparatively easy to see that if such a replacement exists, then the replacement is unique up to isomorphism. Replacements can also be regarded as special instances of the following:

Definition 7.3. An **internal instantiation** of an n-ary metavariable node v in G_1 with an interval (t, b) in G_1 is an equaliser of the following two homomorphisms F and G from the Graph G_0 containing a $(1, n)$-typing element for the label of v:

- F is total and maps the source of G_0 to v,
- G maps the source r_0 of G_0 to t and behaves as b on the successors: $(S_0.r_0);G = b$.

(Such an equaliser is a homomorphism such that $F ; H = G ; H$ and which uniquely factorises any other such homomorphism.) □

Those internal instantiations where the interval has an empty lower border are just the replacements of above.

Definition 7.4. An **external instantiation** of an n-ary metavariable node v in G_1 with an interval (t, b) in G_2 is a pushout of the following two homomorphisms F and G from the Graph G_0 containing a one-source typing element for a metavariable with typing compatible to that of v and with arity $m \leq n$:

– F is total and maps the source of G_0 to v,

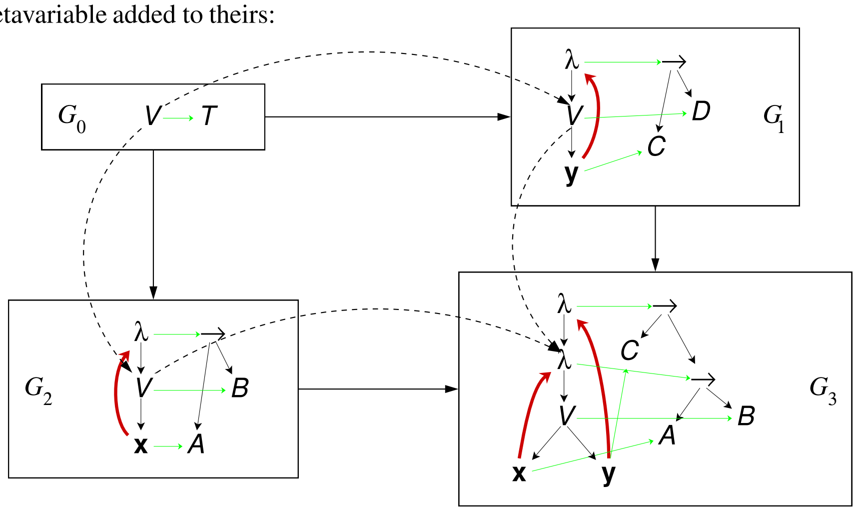
– G maps the source r_0 of G_0 to t and behaves as b on the successors: $(S_0 \cdot r_0) ; G = b$.

(Such a pushout is a "pushout graph" G_3 together with two homomorphisms H_F from G_1 to G_3 and H_G from G_2 to G_3 such that $F ; H_F = G ; H_G$ and H_F and H_G uniquely factorise any other constellation like that.) □

The pushouts of external and the equalisers of internal instantiations need not exist; this fact usually corresponds to the attempt to introduce a type error.

External instantiations with simple intervals and with the metavariable in G_0 having the same arity as the instantiated metavariable can serve to cut or permute outgoing edges of metavariables; another important instantiation uses an interval where one lower border node is identical to the top node, thus "shrinking" every node of the instantiated metavariable to its corresponding successor.

Besides these language-independent instantiations, the most important are those where G_0 contains a zero-ary metavariable mapped to some typing element as G_2; in these cases the metavariables in the typing element have the arity of the instantiated metavariable added to theirs:



Rewriting of our typed term graphs is defined essentially in the same way as in [3,4] via the fibred approach, a generalisation of the traditional double-pushout approach. For reasons of space we do not present this here.

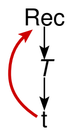
8 Advanced Instances

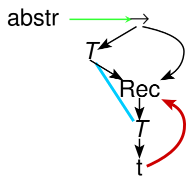
In this section we show how three kinds of advanced type systems can be transferred to the term graph setting. For reasons of space we have to assume that the reader is already familiar with the basics of the respective systems. Although the examples can still be constructed and drawn in HOPS, none of them has been implemented so far because of the problems with second-order unification.

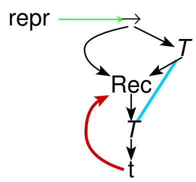
8.1 Recursive Datatypes and Polytypic Programming

The essential feature of polytypic programming — see e.g. [8] — is an explicit recursion operator encapsulating a second-order variable on the type side; this variable stands for some appropriate functor, and the whole construct then stands for the (usually) least fixed point of that functor.

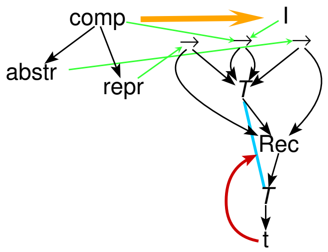
The basic functions necessary for programming on these recursive datatypes are the isomorphisms abstr and repr between the fixed-point domain and its images under the functor. Therefore, we need the following three typing elements — they are all separated:

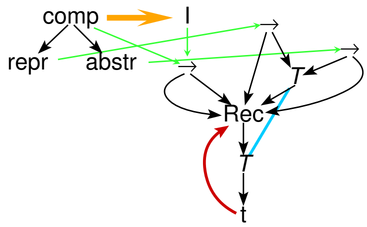






The isomorphism properties can be expressed as rules[1]:



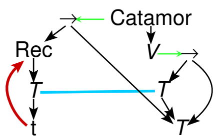


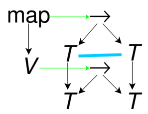
For the first rule, this is indeed the principal type (when limited to maximally identified type parts); the second rule, however, also has another "simplest" type relying on a nested recursion and which is incomparable to the given type — as soon as second-order typing elements are used, automatic unification cannot be used anymore because of these ambiguities of second-order unification. In HOPS, it is planned to make some user-assisted solution available.

---

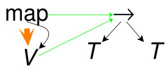
[1] A term graph rule is a term graph together with two distinguished nodes; these two nodes are indicated by thick long-tipped orange resp. grey arrows in HOPS drawings and are the roots of the rule's *left-hand side* and *right-hand side*, respectively. Since rewriting is not the topic of this paper, we do not explain the rule application mechanism here, let us only mention that for multi-node metavariables with successors, no matter whether typed or untyped, the encapsulation skeleton of their image on the left-hand sides has to be copied for constructing their image on the right-hand side.

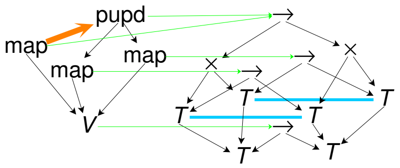
Now, the most popular polytypic functions are probably polytypic `map` and catamorphisms; we give the typing elements:



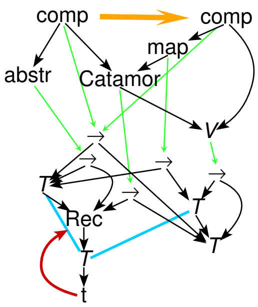


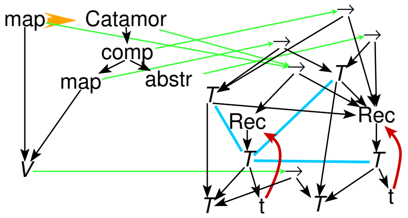
For specific functors, maps can be specialised to known functions, as in the following example rules, which recognise the functor involved by their sensitivity to typing, since their left-hand sides do not have principal types (pupd is — as obvious from its typing — function product):





The basic rule for catamorphisms allows an unfolding of the transferred functor, i.e., a map, to a catamorphism preceded by an abstr; see the drawing to the left:

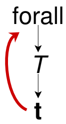


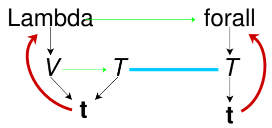


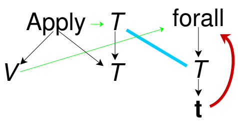
Together with the map rules for the different type constructors — to the right above, there is the map rule for datatype recursion — it is possible to transform arbitrary specific legal catamorphisms into general recursions just by applying these type-sensitive rules.

8.2 Second-order λ-Calculus

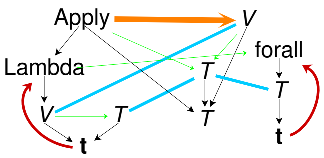
In second-order λ-calculus, there is a different kind of abstraction on the type level, usually denoted "∀"; we shall use the label forall. This then enters the typing elements of a second kind of λ-abstraction and application together with multi-node second-order metavariables:

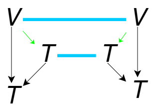






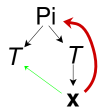
The last two typing elements are not separated: in the abstraction, there is an untyped variable bound at the typed Lambda node, and the typed Apply node has an untyped second successor. The β-reduction rule for this type abstraction nicely shows the interplay between bound variables and metavariables, and with respect to the discussion in Sect. 6 we also show a typing element for the unary typed metavariable used here:

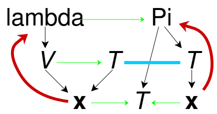


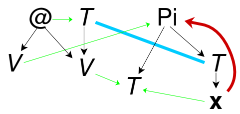


8.3 Dependent Types

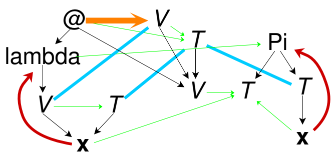
Dependent types are to a certain extent "dual" in a syntactic sense to second-order λ-calculus; here the border crossings between program and type part run in the contrary direction, and there are already border crossings in the typing element for the type constructor Pi: a typed variable x (which has a typing element $\mathsf{x} \longrightarrow T$ of its own) is bound at the untyped Pi, and is successor of an untyped unary metavariable:

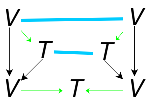






In the typing element of λ-abstraction, the typed bound variable x is also successor to an untyped unary metavariable, and this does not violate the condition that binders should dominate bound variables because of the modified reachability. In the typing element of application, finally, a zero-ary typed metavariable occurs as successor of a unary untyped metavariable — an indication that we have to be very careful in this kind of calculus since now application of rules on the program side may at the same time be on the type side. The pictures of the rule of β-reduction and of the metavariable typing element are astonishingly similar to those for second-order λ-calculus:





9 Conclusion

We have introduced a formalism that allows a large class of type systems to be translated to the term graph setting in a very homogeneous way. The explicitness with which all the structure including typing is incorporated into our term graphs makes them extremely useful for human interaction with complex formalisms — in fact, our formalisation is the result of long-running efforts to provide the graphically interactive term graph programming system HOPS [6,13] with an appropriate typing system both from the implementation point of view and from theoretical considerations.

Another potential use for our internally typed term graphs is to serve as internal data structure in symbolic computation systems including interpreters and compilers, true to the recent trend to keep typing information until much later phases in the compilation process.

References

1. Hendrik P. Barendregt. Lambda Calculi with Types. In S. Abramsky, Dov M. Gabbay, T.S.E. Maibaum, *Handbook of Logic in Computer Science, Vol. 2*, pages 117–309. Oxford Univertity Press, 1992.
2. Chris Brink, Wolfram Kahl, Gunther Schmidt (eds.). *Relational Methods in Computer Science.* Advances in Computing. Springer-Verlag, Wien, New York, 1997. ISBN 3-211-82971-7.
3. Wolfram Kahl. *Algebraische Termgraphersetzung mit gebundenen Variablen.* Reihe Informatik. Herbert Utz Verlag, München, 1996. ISBN 3-931327-60-4, zugleich Dissertation an der Fakultät für Informatik, Universität der Bundeswehr München.
4. Wolfram Kahl. A Fibred Approach to Rewriting — How the Duality between Adding and Deleting Cooperates with the Difference between Matching and Rewriting. Tech. Rep. 9702 (May 1997), Fakultät für Informatik, Universität der Bundeswehr München.
5. Wolfram Kahl. Relational Treatment of Term Graphs With Bound Variables. *Journal of the IGPL* **6** (2), 259–303 (March 1998).
6. Wolfram Kahl. The **H**igher **O**bject **P**rogramming **S**ystem — User Manual for HOPS, Fakultät für Informatik, Universität der Bundeswehr München, February 1998. URL http://diogenes.informatik.unibw-muenchen.de:8080/kahl/HOPS/.
7. Jan Willem Klop. Combinatory Reduction Systems. Mathematical Centre Tracts 127 (1980), Centre for Mathematics and Computer Science, Amsterdam. PhD Thesis
8. Erik Meijer, Maarten Fokkinga, Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In John Hughes (ed.), *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, pages 124–144. LNCS 523. Springer Verlag, 1991.
9. Gunther Schmidt, Thomas Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists.* EATCS-Monographs on Theoretical Computer Science. Springer-Verlag, Berlin/Heidelberg/New York, 1993.
10. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.
11. Carolyn L. Talcott. A Theory of Binding Structures and Applications to Rewriting. *Theoretical Computer Science* **112**, 68–81 (1993).
12. Christopher Peter Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* Ph.D. thesis, Oxford University, September 1971.
13. Hans Zierer, Gunther Schmidt, Rudolf Berghammer. An Interactive Graphical Manipulation System for Higher Objects Based on Relational Algebra. In *Proc. 12th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 68–81. LNCS 246. Springer-Verlag, Bernried, Starnberger See, June 1986.