Consider a crude style rule basis Σ that is concrete (no polymorphism) and complete (each type has a rule). Build the final type-graph Δ, whose nodes are the concrete type symbols, and whose arcs are given by: for each ξ, if $\xi \leftarrow \zeta_1 \ldots \zeta_n$ is the rule for ξ, then the k'th child of ξ is ζ_k. If a system **R** is well typed acording to Σ and a suitable B, then for every execution graph G of the system, there is a node-symbol-forgetting homomorphism $h : G \to \Delta$. Readers should convince themselves that this property fails for the imperative discipline.

To conclude then, we have discussed type inference for TGRSs and shown that credible type systems can be built using dataflow analysis and unification, even in the most general case where no particular nice structural properties are assumed for the system, fulfilling the promise in the conclusion of Banach (1989). We have also suggested that more powerful extensions of these systems can be contemplated. These extensions will be described elsewhere.

References

Banach R. (1989), Dataflow Analysis of Term Graph Rewriting Systems, *in* proc. PARLE-89, Odijk E., Rem M., Syre J-C. *eds*., Springer, LNCS **366** 55-72.

Banach R. (1991a), DACTL Rewriting is Categorical, *in* proc. SemaGraph-91, University of Nijmegen Dept. of Informatics Technical Report **91-25** part II 339-357. Also see *in*: Term Graph Rewriting: Theory and Practice, John Wiley, 1992, *to appear.*

Banach R. (1991b), Term Graph Rewriting and Garbage Collection à la Grothendieck. *Submitted to TCS.*

Banach R. (1991c), MONSTR: Term Graph Rewriting for Parallel Machines, *in* proc. SemaGraph-91, University of Nijmegen Dept. of Informatics Technical Report **91-25** part II 251-260. Also see *in*: Term Graph Rewriting: Theory and Practice, John Wiley, 1992, *to appear.*

Banach R. (1992), MONSTR, *in preparation.*

Barendregt H.P. (1984), The Lambda Calculus. Its Syntax and Semantics, North-Holland.

Barendregt H.P., van Eekelen M.C.J.D., Glauert J.R.W., Kennaway J.R., Plasmeijer M.J., Sleep M.R. (1987), Term Graph Rewriting, *in* proc. PARLE-87, de Bakker J.W., Nijman A.J., Treleaven P.C. *eds*., Springer, LNCS **259** 141-158.

Farmer W.M., Watro R.J. (1990), Redex Capturing in Term Graph Rewriting, Int. Jour. Found. Comp. Sci. **1** 369-386, and *in* proc. RTA-91, R.V. Book *ed*., Springer, LNCS **488** 13-24.

Girard J-Y., Taylor P., Lafont Y. (1989), Proofs and Types, Cambridge Tracts in Theoretical Computer Science 7, CUP.

Glauert J.R.W., Kennaway J.R., Sleep M.R., Somner G.W. (1988a), Final Specification of DACTL, Internal Report SYS-C88-11, School of Information Systems, University of East Anglia, Norwich, U.K.

Glauert J.R.W., Hammond K., Kennaway J.R., Papdopoulos G.A., Sleep M.R. (1988b), DACTL: Some Introductory Papers, School of Information Systems, University of East Anglia, Norwich, U.K.

Hankin C. (1991), Static Analysis of Term Graph Rewriting Systems, *in* proc. PARLE-91, Aarts E.H.L., van Leeuwen J., Rem M. *eds*., Springer, LNCS **506** 367-384.

Hindley R. (1969), The Principal Type-Scheme of an Object in Combinatory Logic, Trans. Amer. Math. Soc. **146** 29-60.

Huet G. (1990), Logical Foundations of Functional Programming, Addison-Wesley.

Kennaway J.R., Klop J-W., Sleep M.R., de Vries F-J. (1991), Transfinite Reductions in Orthogonal Term Rewrite Systems, *in* proc. RTA-91, R.V. Book *ed*., Springer, LNCS **488** 1-12, and Report CS-R9041, CWI Amsterdam.

Milner R. (1978), A Theory of Type Polymorphism in Programming, Jour. Comp. Sys. Sci. **17** 348-375.

Milner R., Tofte M. (1991), Co-induction in Relational Semantics, Theor. Comp. Sci. **87** 209-220.

Peyton-Jones S.L. (1987), The Implementation of Functional Programming Languages, Prentice-Hall.

Tofte M. (1990), Type Inference for Polymorphic References, Inf. and Comp. **89** 1-34.

(1969), Milner (1978)), whose use of unification inspired our use of it here. The main difference between ours and the H-M system is of course the absence of structural induction and its replacement by dataflow analysis. This global analysis feature is somewhat reminiscent of the Milner-Tofte (M-T) type system for ML with assignments[1] (Milner and Tofte (1991), Tofte (1990)), though we have not used co-induction in the explicit way that they do. Moreover for us, even the patterns occurring in rules, which are the templates for the objects we want to type i.e. the execution graphs, can be cyclic, and fail to be freely generated by recursion. This is of course fundamental and is the main impetus for bringing in the dataflow analysis.

Turning now to the types themselves, the types in our system fail to have anything but a trivial structure, unlike the H-M case where there are typically formation rules such as

$$\frac{\alpha \text{ is a type} \quad \beta \text{ is a type}}{\alpha \times \beta \text{ is a type}} \qquad\qquad \frac{\alpha \text{ is a type} \quad \beta \text{ is a type}}{\alpha \to \beta \text{ is a type}}$$

(it might indeed have been more honest to call our system sort inference rather than type inference). However this is something we can emulate easily enough. Let us take an H-M style language of types **TL** given by the syntax

$$\alpha \in \mathbf{TL} = b \mid v \mid \alpha_1 \to \alpha_2 \mid \alpha_1 \times \alpha_2$$

where b represents base types (*Int*, *Bool* …) and v represents variables. Let $\lceil\ \rceil$ be a Gödelisation of **TL** that maps each $\alpha \in \mathbf{TL}$ to its "Gödel number" $\lceil\alpha\rceil$ in $\mathbf{Z}^c$. With a suitable reinterpretation of the unification steps in algorithm 5.3 involving the Gödel number coding of unification in **TL**, algorithm 5.3 will do duty as a correctness checker/type inferer for the new scheme; the occurs check maintaining the termination properties of 5.3. We have thus another separation of concerns, the separation of unification in **TL** from other aspects of our system. Let us briefly look at how this emulation works. There are two basic styles we can use, the functional and the applicative. The functional style is similar to what we are used to already. Say we wish to apply an operator $\mathsf{F}^{\alpha \times \beta \to \gamma}$ to operands A^{α} and B^{β}, giving $(\mathsf{F}^{\alpha \times \beta \to \gamma}(\mathsf{A}^{\alpha}, \mathsf{B}^{\beta}))^{\gamma}$. We can emulate this H-M situation by having $\{\mathsf{F}^{\lceil\gamma\rceil}, \mathsf{A}^{\lceil\alpha\rceil}, \mathsf{B}^{\lceil\beta\rceil}\} \subseteq B$, and a rule $(\mathsf{F}, \lceil\gamma\rceil) \leftarrow \lceil\alpha\rceil, \lceil\beta\rceil \in \Sigma$. On the other hand, the applicative style introduces special purpose application symbols **App***n* (of arity $\{1 \ldots n\}$) to represent the explicit application of an operator to its arguments. With the previous $\mathsf{F}^{\alpha \times \beta \to \gamma}$, A^{α}, B^{β}, our little example becomes $(\mathbf{App}3(\mathsf{F}^{\alpha \times \beta \to \gamma}, \mathsf{A}^{\alpha}, \mathsf{B}^{\beta}))^{\gamma}$. Our emulation now requires $\{\mathsf{F}^{\lceil\alpha \times \beta \to \gamma\rceil}, \mathsf{A}^{\lceil\alpha\rceil}, \mathsf{B}^{\lceil\beta\rceil}, \mathbf{App}3^{\lceil\gamma\rceil}\} \subseteq B$, and $(\mathbf{App}3, \lceil\gamma\rceil) \leftarrow \lceil\alpha \times \beta \to \gamma\rceil, \lceil\alpha\rceil, \lceil\beta\rceil \in \Sigma$. Note that in this case we really would need to overload the **App**3 symbol, since we would have to assume that there were other operators such as $\mathsf{G}^{\alpha' \times \beta' \to \gamma'}$ of type different from F, which would also use the **App**3 combinator. This in turn would require us to at least use the finegrained version of dataflow analysis to have a nontrivial system. In fact all the rules of the system would be instantiations of

$$(\mathbf{App}n, \lceil v_n\rceil) \leftarrow \lceil v_1 \times v_2 \times \ldots \times v_{n-1} \to v_n\rceil, \lceil v_1\rceil, \lceil v_2\rceil \ldots \lceil v_{n-1}\rceil$$

where n is a meta-variable and the v_i's are variables of **TL** (note that any non-ground instantiation of the above is non-linear modulo Gödel numbering). In such a case a strategy for for typechecking that just enforced the consistency condition on the children of an **App***n* node and infered the type of the parent, rather than concentrating on the parent-child type relationship, would work more simply. Modifying algorithm 5.3 to do this is not too hard.

One aspect of H-M and M-T systems definitely missing from ours is the necessity to deal with bound variables and non-trivial environments created by static scoping constructs such as (**let** $x = e$ **in** e') and lambda abstractions. The absence of these features, attributable to the "flatness" of TGRSs affords us some considerable simplification.

In case the reader by now thinks that the crude discipline has no merits not amply exceeded by the imperative discipline, we point out one *final* property not shared by its imperative brother or his fancier cousins.

1. The author is indebted to Kris Rose for bringing the Milner-Tofte work to his attention, and he intends to explore the connection between the techniques used here and those of the M-T system elsewhere.

Let us start with the crude discipline and let (Σ, B) be given by $(\Sigma_1 \cup \{\mathit{Int} \leftarrow \mathit{Int}, \mathit{Int}\}, B_1 \cup \{\mathsf{Inc}^{\mathit{Up}}, \mathsf{Plus}^{\mathit{Int}}, \mathsf{Count}^{C}\})$ where (Σ_1, B_1) refer to the other parts of the system. Since $\mathsf{Inc}' \in R(\mathsf{Inc})$, $\mathsf{Inc}'\ \rho^0_s\ \mathsf{Inc}$ whence $\mathsf{Inc}'^{\mathit{Up}}$ is an SDLPTI typing. Similarly Count'^{C}. We see also that if there were a monomorphic type rule for C, it would have to be $C \leftarrow \mathit{Int}$ because of the instance of the germ $(\mathsf{Count}'_1, \mathsf{Plus})$ in the contractum of P. From $\mathsf{Plus}^{\mathit{Int}}$ and $\mathit{Int} \leftarrow \mathit{Int}, \mathit{Int}$ we deduce that $\mathsf{a} : \mathsf{Any}^{\mathit{Int}}$ and $\mathsf{l} : 1^{\mathit{Int}}$. The former is consistent with the $C \leftarrow \mathit{Int}$ we discovered above and tells us that any node matched to the particular implicit node a in a redex for our rule must be of type Int if the system is to be SDLPTI-typeable. The latter spells trouble. From $\mathit{Int} \leftarrow \mathit{Int}, \mathit{Int}$ and 1^{Int} we deduce that $A(1) = \{1\ldots2\}$. But this is nonsense since $\mathsf{l} : 1$ has no children in P and l is an explicit node. This means that any system including our rule is not SDLPTI-typeable using the crude discipline and given (Σ, B). We therefore see that the crude discipline is very severe in restricting the arities of similarly typed symbols to be the same.

Now let us examine the same example using the simple imperative discipline and let (Σ, B) be $(\Sigma_1 \cup \{(\mathsf{Plus}, \mathit{Int}) \leftarrow \mathit{Int}, \mathit{Int}, (1, \mathit{Int}) \leftarrow \}, B_1 \cup \{\mathsf{Inc}^{\mathit{Int}}, \mathsf{Count}^{C}, \mathsf{Plus}^{\mathit{Int}}, 1^{\mathit{Int}}\})$. Note that since the types of child nodes now depend on the parent type and parent symbol rather than just the parent type alone, the previous problem does not arise and we can even safely have $\mathsf{Inc}^{\mathit{Int}}$ in B. On the down side, we have to put more into the basis because of the weaker constraints enforced. Much as before, we can conclude $\mathsf{Inc}'^{\mathit{Int}}$, Count'^{C} and $\mathsf{a} : \mathsf{Any}^{\mathit{Int}}$, and can consistently postulate the type rules $(\mathsf{Inc}, \mathit{Int}) \leftarrow C$, $(\mathsf{Inc}', \mathit{Int}) \leftarrow C$, $(\mathsf{Count}, C) \leftarrow \mathit{Int}$, $(\mathsf{Count}', C) \leftarrow \mathit{Int}$.

So the imperative discipline allows much more natural looking rules to be well typed. One can see the similarity with imperative languages, where operators are endowed with a result type and sequence of operand types, and type checking proceeds by matching the type at a point in the program against the type demanded for a child at the relevant position of its parent (as in Pascal); or more generally some inference is done (as in Ada).

Note also that the presentation of type rules as $(S, \xi^c) \leftarrow \zeta_1\ldots\zeta_n \in \mathbf{S} \times \mathbf{Z}^c \to \mathbf{Z}^*$ apparently allows overloading by permitting

$$(S, \xi^c_1) \leftarrow \zeta_1\ldots\zeta_n \text{ and } (S, \xi^c_2) \leftarrow \zeta'_1\ldots\zeta'_n \quad (\text{with } \xi^c_1 \neq \xi^c_2)$$

to coexist in Σ. This is true, but useless in the context of this paper since the first thing that algorithm 5.3 does is to attempt to unify the types of all occurrences of S in step [5] so at most one of the above rules would ever be used. To exploit potential overloading we need to re-engineer the dataflow analysis with a go-faster supercharged version, capable of discriminating different occurrences of the same node symbol in some way. A more finegrained version of dataflow analysis can do this but is beyond the scope of this paper. (In actual fact it would inevitably go slower rather than faster.)

The main reason we used the more relational presentation for rules, i.e. $\mathbf{S} \times \mathbf{Z}^c \to \mathbf{Z}^*$ rather than the functional $\mathbf{S} \to \mathbf{Z}^c \to \mathbf{Z}^*$, is the uniformity of presentation of the crude and imperative disciplines it allows. (We note that a fully relational $\mathbf{S} \times \mathbf{Z}^c \times \mathbf{Z}^*$ for the rule basis, and $\mathbf{S} \times \mathbf{Z}^c$ for the axiom basis could have been contemplated.) Presenting crude rules in the form $\{\bullet\} \to \mathbf{Z}^c \to \mathbf{Z}^*$ would clearly not have worked. As it is, (very nearly) the same theory will do for both disciplines, useful for comparison and for saving space. This also neatly illustrates the nice separation of concerns achieved by making the inference engine of the type discipline (dataflow analysis), independent of the correctness checker (unification). Either or both components may be traded up for a more powerful model relatively independently. (We ruefully observe that the first thing that the design of an efficient implementation of either discipline would do, would be to jam the loops implicit in dataflow analysis and unification, wrecking this independence.)

Now for some more general remarks. As with most useful type systems, typability of systems in the dynamic sense of 4.1.2 or 4.2.2 is undecidable (it is easy to reduce the halting problem to it); thus the SDLPTI algorithm provides an intentionally based approximation to typability. The algorithm reminds us of some features of other type systems. Prominent among these is the Hindley-Milner (H-M) system (Hindley

Proof. If the SDLPTI algorithm SUCCEEDs, then we know we have a correct typing provided we can be sure that the extra condition implicit in 4.5.(2) for non-linear rules holds, namely that if $\Sigma(\xi)[k_1] = \zeta^{\mathrm{v}} = \Sigma(\xi)[k_2]$ for $k_1 \neq k_2$, then any node of type ξ has children of identical type at its k_1 and k_2'th positions.

From the properties of dataflow analysis and the SDLPTI algorithm we know, as remarked previously, that if (p_k, c) is some arc of an execution graph G_i, then for all G_j with $j > i$, $i_{G_i,G_j}(p)$ is of the same type as p, and $r_{G_i,G_j}(c)$ is of the same type as c. So to check the property required, we need only check it at the points at which contractum nodes x are instantiated as nodes of execution graphs. But at all such points we have the structure of the rule governing the rewrite to help us, because it gives direct information about the children of x and hence about the children of the instantiation of x. If such a child is explicit, its symbol can be read off from the rule and its type determined (being the SDLPTI type), thus giving the type of the child of the instantiation. This corresponds to case (1) above. If the child is implicit however, we must refer to the set $O_k(\sigma(x)^{\xi^{\mathrm{c}}})$. This may contain many symbols, but if it happens to be a type singleton, we can likewise determine the type of the child of the instantiation. This corresponds to case (2) above. Thus under the given hypotheses, the uniformity of the types of the children across the relevant set of positions of x gives us the equality we need for the extra condition. ☺

The above theorem has slightly different implications depending on whether the θ for a given $W_{\xi^{\mathrm{c}}}(\zeta^{\mathrm{v}})$ is concrete or not. If it is concrete, then it clearly tells us the type of the relevant children of (instantiations of) x explicitly. On the other hand, if it is a type variable, then it merely tells us that it is *possible* to type the system consistently, without offering us an explicit type for the children in question. In such a case (as in others in which the SDLPTI type of some symbols are type variables) the type structure we started with is too weak to fully type the system. Only a less general unifier would concretely type the system, and for that we need some more axioms and/or some more rules. In any event, the success of the SDLPTI algorithm assures us that such a consistent extension of the type structure exists.

Note that the power of the above result for dealing with genuine non-linear polymorphism comes from the dependence of θ on x. This in turn only has any real force when condition (2) is never needed, i.e. when all relevant children of contractum nodes labelled by symbols of type ξ^{c} are explicit, freeing the dependence of θ upon x from $O_k(\sigma(x))$ which for a fixed symbol $\sigma(x)$, does not depend on the node x that happens to bear that symbol. Contrast this with the linear case where $O_k(\sigma(x))$ does not have to be a type singleton and can thus contain several differently typed $R(–)$ sets, this being the chief means by which polymorphism is achieved within our TGR framework.

6 Discussion and Conclusions

We have defined TGRSs, dataflow analysis, type structures, and we have shown how the properties of dataflow analysis are suited to doing type inference of the kind required by our particular brand of type theory. Some points are worthy of further discussion, so let's look at an example and compare the present framework to conventional ones.

Example 6.1 We'll continue our slightly half-hearted example introduced previously, constrained as we inevitably are by not having given a complete system. Again we'll refer to the P shown in section 2. Now our definitions for correctness made no mention of the typing of rules. This was deliberate, since the semantics of the variable Any nodes in a pattern (by which we mean the attributes of the execution graph nodes that they will match) are entirely dependent on the rest of the system, and trying to assign types to them cannot be contemplated until the existence of a correct typing for the whole system has been established. In this sense, talking of the typing of P is contingent on such a global typing. Assuming such, the dataflow analysis of example 3.3 and the properties of the unification algorithm 5.3 allow us to infer certain things about symbols and germ instances occuring in P.

accomplished by steps [1] – [5], which build the equivalence classes of ρ_S and perform the unification with the axiom basis.

It remains to unify the resulting type labelling of symbols with the rule basis Σ. This is accomplished by the loop [8i] – [12i]. To enforce conformance to Σ we have to check relationships between type symbols at parent and child ends of all potential arcs of execution graphs. All such arcs are instances of germs that can be constructed from the $O_k(–)$ sets. So let $(S^{\xi}{}_k, T^{\zeta})$ be such a germ where the symbols bear the type symbols appropriate to the i'th iteration of the algorithm. If $\xi \in \mathbf{Z}^{\mathrm{v}}$ there is nothing to check. Otherwise if $\xi \in \mathbf{Z}^{\mathrm{c}}$ and $\Sigma(S, \xi)$ is defined, then arities must match, and if $\Sigma(S, \xi)[\mathrm{k}] \in \mathbf{Z}^{\mathrm{c}}$, then if $\zeta \in \mathbf{Z}^{\mathrm{c}}$ too, then ζ must $= \Sigma(S, \xi)[\mathrm{k}]$ (else FAILure). If $\Sigma(S, \xi)[\mathrm{k}] \in \mathbf{Z}^{\mathrm{c}}$ but $\zeta \in \mathbf{Z}^{\mathrm{v}}$, then ζ must be instantiated to $\Sigma(S, \xi)[\mathrm{k}]$. There may be several such competing instantiations for ζ arising from other germs $(U^{\chi}{}_l, T^{\zeta})$. Any one such instantiation will do, since if they all agree (i.e. all wish to map ζ to the same $\theta^{\mathrm{c}} \in \mathbf{Z}^{\mathrm{c}}$) then there is no conflict, while if they disagree, FAILure is invevitable at the i+1'th iteration. This explains the form of the else branch of [10i].

So the loop progressively instantiates type variables, only when needed, until termination or FAILure. It clearly generates a most general unification of Σ with t-$\mathbf{S}^0$, hence a most general unification of t-$\mathbf{S}$ with (Σ, B). Thus t-$\mathbf{S}^f$ is a correct typing of $\mathbf{R}$. ☺

It goes without saying that the above algorithm, is not one that would be implemented as given, being constructed primarily for readability. Many aspects can be optimised to produce a practical algorithm, but these issues are outside the scope of this paper.

As promised, we will strengthen theorem 5.4 to enable the SDLPTI algorithm to effectively generate correct typings in certain non-linear cases. The problem with non-linearity is that if $k_1 \neq k_2$ and $\Sigma(S, \xi^{\mathrm{c}})[k_1] = \zeta^{\mathrm{v}} = \Sigma(S, \xi^{\mathrm{c}})[k_2]$ then if a graph node x is of type ξ^{c} then its k_1 and k_2'th children must be of the same type, θ_1 say, even if that type is different to the type, θ_2 say, of the k_1 and k_2'th children of y, where y is also of type ξ^{c}. Since dataflow analysis and hence the SDLPTI algorithm works on each germ (and perforce each germ instance) independently, they are unable to distinguish this case from the case where the k_1'th child of x is of type θ_1, the k_2'th child of x is of type θ_2, the k_1'th child of y is of type θ_1, and the k_2'th child of y is of type θ_2. The latter is a non-typing of the graph containing x and y according to 4.5.

However, if we can be sure that every node of type ξ^{c} created during a rewrite has its k_1 and k_2'th children of equal type at the moment of creation, then we can use the SDLPTI algorithm to check that the system is correctly typed. This is because the SDLPTI algorithm ensures that the functions $i_{G,H}(–)$, $r_{G,H}(–)$ preserve type, so if the children are of equal type at the point the parent is created, then they remain of equal type subsequently, despite redirection. Clearly this is an observation outside the remit of the standard SDLPTI algorithm.

Theorem 5.5 Let $\mathbf{S}$, $\mathbf{Z}$, $\mathbf{R}$ and (Σ, B) be as before, but suppose (Σ, B) is non-linear. For $\xi^{\mathrm{c}} \in \mathbf{Z}^{\mathrm{c}}$ and $\zeta^{\mathrm{v}} \in \mathbf{Z}^{\mathrm{v}}$, let $W_{\xi^{\mathrm{c}}}(\zeta^{\mathrm{v}})$ be the set of indices $W_{\xi^{\mathrm{c}}}(\zeta^{\mathrm{v}}) = \{k \mid \Sigma(S, \xi^{\mathrm{c}})[k] = \zeta^{\mathrm{v}}\}$. Suppose for all (S, ξ^{c}) and ζ^{v} such that $\Sigma(S, \xi^{\mathrm{c}})$ is defined and non-linear and $W_{\xi^{\mathrm{c}}}(\zeta^{\mathrm{v}})$ is a non-singleton, and for all x a contractum node of the pattern P of a rule D of $\mathbf{R}$, with $\sigma(x) = S$ and ξ^{c} the SDLPTI type of S, there is a $\theta \in \mathbf{Z}$ (depending on both x and $W_{\xi^{\mathrm{c}}}(\zeta^{\mathrm{v}})$), such that for all $k \in W_{\xi^{\mathrm{c}}}(\zeta^{\mathrm{v}})$, either

(1) The k'th child of x is an explicit node y_k, and θ is the SDLPTI type of $\sigma(y_k)$, or

(2) The k'th child of x is an implicit node, and $O_k(\sigma(x)^{\xi^{\mathrm{c}}})^{\tau} = \{\theta\}$ (i.e. a singleton SDLPTI type).

Then the output of the SDLPTI algorithm is a correct typing of the system.

```
[6]    Let t-$\mathbf{S}^0$ be given by $S^\xi \in$ t-$\mathbf{S} \Leftrightarrow S^{I^0(\xi)} \in$ t-$\mathbf{S}^0$.  Let $\rho^0$ be given by $T^\zeta\ \rho\ S^\xi \Leftrightarrow T^{I^0(\zeta)}\ \rho^0\ S^{I^0(\xi)}$.
       Let $\rho^0_s$ be the symmetric closure of $\rho^0$ and write $[S^\xi]^0$ for a typical component.  Let t-$O^0_k(-)$
       sets be given by $T^\zeta \in$ t-$O_k(S^\xi) \Leftrightarrow T^{I^0(\zeta)} \in$ t-$O^0_k(S^{I^0(\xi)})$.
[7]    Let  $i = 0$.
[8i]   **Repeat**
[9i]   $i := i + 1$
[10i]  **If**  for any $S^\xi, T^\zeta \in$ t-$\mathbf{S}^{i-1}$ we have $\xi \in \mathbf{Z}^c$ and $\Sigma$ contains a rule for $\underline{S \text{ and}}\ \xi$ such
          that $A(\underline{S,}\ \xi) \neq A(S)$, or there is a $k \in A(S, \xi)$ such that $\Sigma(S, \xi)[k] = \theta^c \in \mathbf{Z}^c$ and
          $T^\zeta \in$ t-$O^{i-1}_k(S^\xi)$ with $\theta^c \neq \zeta \in \mathbf{Z}^c$
       **Then**  FAIL and exit
       **Else**  Define a substitution $I^i$ by
              **For**  all $S^\xi, T^\zeta \in$ t-$\mathbf{S}^{i-1}$  **Do**
                     **If**  $\zeta \in \mathbf{Z}^v, \xi \in \mathbf{Z}^c$ and $\Sigma$ contains a rule for $\underline{S \text{ and}}\ \xi$ such that there is a
                        $k \in A(\underline{S,}\ \xi)$ such that $T^\zeta \in$ t-$O^{i-1}_k(S^\xi)$
                     **Then**  $I^i(\zeta) = \theta^c$ where $\theta^c$ is any such $\xi$
              **For**  all remaining $\theta \in \mathbf{Z}^v$, $I^i(\theta) = \theta$
[11i]  Let t-$\mathbf{S}^i$ be given by $S^\xi \in$ t-$\mathbf{S}^{i-1} \Leftrightarrow S^{I^i(\xi)} \in$ t-$\mathbf{S}^i$.  Let $\rho^i$ be given by $T^\zeta\ \rho^{i-1}\ S^\xi \Leftrightarrow T^{I^i(\zeta)}\ \rho^i\ S^{I^i(\xi)}$.
       Let $\rho^i_s$ be the symmetric closure of $\rho^i$  and write $[S^\xi]^i$ for a typical component.  Let t-$O^i_k(-)$
       sets be given by $T^\zeta \in$ t-$O^{i-1}_k(S^\xi) \Leftrightarrow T^{I^i(\zeta)} \in$ t-$O^i_k(S^{I^i(\xi)})$.
[12i]  **Until**  $I^i = \mathrm{Id}_{\mathbf{Z}}$
[13]   Let $f$ be the final value for $i$.  Output t-$\mathbf{S}^f$ and SUCCEED.
```

As a matter of teminology, if the SDLPTI algorithm SUCCEEDs on some system, then for every $S \in \mathbf{S}$, if $S^\xi \in$ t-$\mathbf{S}^f$ where t-$\mathbf{S}^f$ is the set output by the algorithm, we call ξ the **SDLPTI type** of S.

Theorem 5.4 Let $\mathbf{S}$, $\mathbf{Z}$, $\mathbf{R}$, (Σ, B) be given as before with $\mathbf{S}$ and $\mathbf{R}$ finite and membership of $\mathbf{Z}$ and (Σ, B) recursively decidable. Suppose (Σ, B) is linear.

(1) The SDLPTI algorithm terminates and either SUCCEEDs or FAILs.

(2) If it SUCCEEDs, then $\mathbf{R}$ is correctly typed by (Σ, B) and a suitable typing of any execution graph may be given by typing each node x of the graph by the SDLPTI type of $\sigma(x)$.

Proof Because $\mathbf{S}$ and $\mathbf{R}$ are finite and all the substitutions mentioned have finite support, all the individual steps of the algorithm are finitely computable. To show that there are a finite number of iterations of steps [9i] – [12i] we note that each iteration except the last retypes a finite non-zero number of symbols decorated with variables, to symbols decorated with constants. Since no symbol decorated with a constant is ever retyped, and there are only a finite number of symbols involved to start with, the loop must terminate. The only exit points of the algorithm are when it SUCCEEDs or FAILs, so we have (1).

We note that by 3.2.(R), the ρ^i relations are reflexively and transitively closed, so their symmetric closures are indeed equivalence relations.

As to the output when the algorithm SUCCEEDs, we need only show that the three conditions of theorem 5.2 hold with respect to the set of t-symbols t-$\mathbf{S}^f$, created by the algorithm.

To satisfy 5.2.(3), we must ensure that all members of an $R(-)$ set have the same type. Similarly, to satisfy 5.2.(1), we must ensure any symbol typed by the axiom basis B, has the same type in t-$\mathbf{S}^f$. Both tasks are

Theorem 5.2 Suppose (Σ, B) is linear. Suppose all symbols in $\mathbf{S}$ are decorated with a type, yielding a set of t-symbols t-$\mathbf{S}$. For each t-symbol, let $R(–)$ and $O_k(–)$ sets of t-symbols be given that satisfy the hypotheses of theorem 3.2 (where we ignore the type decorations for the purposes of 3.2). Suppose also for each $S^\xi \in$ t-$\mathbf{S}$,

(1) If $B(S)$ is defined, then $B(S) = \xi$,

(2) If $\Sigma(S, \xi)$ is defined, then $A(S) = A(S, \xi)$, and for each $k \in A(S, \xi)$ such that $\Sigma(S, \xi)[k] = \theta^c \in \mathbf{Z}^c$, for all $T^\zeta \in O_k(S^\xi)$, $\zeta = \theta^c$,

(3) $R(S^\xi)$ is a type singleton.

Then (Σ, B) correctly types $\mathbf{R}$ when each execution graph node x is typed by θ where $\sigma(x)^\theta$ is in t-$\mathbf{S}$.

Proof. Let G_i be an execution graph and $x \in G_i$. Map all nodes to type symbols as suggested. Then (1) implies (1) of 4.2.1. By theorem 3.2, if y is the k'th child of x, then $\sigma(y) \in O_k(\sigma(x))$, and thus (2) trivially guarantees (2) of 4.2.1 because (Σ, B) is linear. So the map is indeed a typing of G_i. Theorem 3.2 also guarantees that for all $j > i$, $\sigma(r_{G_i,G_j}(x)) \in R(\sigma(x))$, hence (3) ensures that we have a correct typing of the system. ☺

We have suggested that for dataflow analysis, suitable $R(–)$ and $O_k(–)$ sets may be obtained by iteration. We now present an algorithm for determining suitable sets of t-symbols t-$\mathbf{S}$ for which the conditions of theorem 5.2 hold.

Algorithm 5.3 (The SDLPTI algorithm)

[1] Decorate each $S \in \mathbf{S}$ with a fresh type variable from $\mathbf{Z}^v$ which does not occur in Σ. Call the resulting set of t-symbols t-$\mathbf{S}$.

[2] Let t-$R(–)$ and t-$O_k(–)$ sets be given by

$$T^\zeta \in \text{t-}R(S^\xi) \Leftrightarrow S^\xi, T^\zeta \in \text{t-}\mathbf{S} \text{ and } T \in R(S)$$
$$T^\zeta \in \text{t-}O_k(S^\xi) \Leftrightarrow S^\xi, T^\zeta \in \text{t-}\mathbf{S} \text{ and } T \in O_k(S)$$

[3] Define ρ on t-$\mathbf{S}$ by

$$T^\zeta \,\rho\, S^\xi \Leftrightarrow T^\zeta \in \text{t-}R(S^\xi)$$

and let ρ_s be the symmetric closure of ρ. Then ρ_s is an equivalence relation. Write $[S^\xi]$ for the component of ρ_s containing S^ξ.

[4] For each $[S^\xi]$ let

$$[S^\xi]_B = [S^\xi] \cup \{T^{\zeta^c} \mid B(T) = \zeta^c \text{ and } T^{\theta^v} \in [S^\xi] \text{ for some } \theta^v \in \mathbf{Z}^v\}$$

[5] Unify each $[S^\xi]_B^\tau$. That is

```
If any $[S^\xi]_B$ is not acceptable Then FAIL and exit
Else Define a substitution $I^0$ by
    For all $[S^\xi]_B$ Do
        If $[S^\xi]_B^\tau$ contains a concrete type symbol $\theta^c$
        Then For all $\zeta \in \mathbf{Z}^v \cap [S^\xi]_B^\tau$, let $I^0(\zeta) = \theta^c$
        Else Choose some $\theta^v \in [S^\xi]_B^\tau$ and for all $\zeta \in \mathbf{Z}^v \cap [S^\xi]_B^\tau$, let $I^0(\zeta) = \theta^v$
    For all remaining $\theta \in \mathbf{Z}^v$, $I^0(\theta) = \theta$
```

$$\tau(\alpha(x)[k]) = s_x(\Sigma(\tau(x))[\mathrm{k}]).$$

We say that τ is a **concrete typing** of G iff $\tau(G) \subseteq \mathbf{Z}^{\mathrm{c}}$.

Definition 4.1.2 Let typings be given for all execution graphs and let the following hold for all executions of **R**. If G_i, G_j are in some execution with $i < j$, and τ_i, τ_j are the given typings, then for all $x \in G_i$

$$\tau_i(x) = \tau_j(i_{G_i,G_j}(x)) = \tau_j(r_{G_i,G_j}(x))$$

Then we say that **R** is **correctly typed** by (Σ, B). If all such typings are concrete, we say that **R** is **concretely correctly typed** by (Σ, B).

4.2 A Simple Imperative Type Discipline

Let Ω be **S** the node symbol alphabet, and let some TGRS **R**, and some type theory (Σ, B) be considered fixed.

Definition 4.2.1 Let G be a graph and let $\tau : G \to \mathbf{Z}$ be a map from the nodes of G to type symbols. Then τ is a **typing** of G iff for all $x \in G$,

(1) If $B(\sigma(x))$ is defined, then $\tau(x) = B(\sigma(x))$.

(2) If $\Sigma(\sigma(x), \tau(x))$ is defined, then $A(\sigma(x), \tau(x)) = A(\sigma(x)) = A(x)$ and there is a(n x-dependent) substitution s_x such that for all $k \in A(x)$,

$$\tau(\alpha(x)[k]) = s_x(\Sigma(\sigma(x), \tau(x))[\mathrm{k}]).$$

We say that τ is a **concrete typing** of G iff $\tau(G) \subseteq \mathbf{Z}^{\mathrm{c}}$.

Definition 4.2.2 Let typings be given for all execution graphs and let the following hold for all executions of **R**. If G_i, G_j are in some execution with $i < j$, and τ_i, τ_j are the given typings, then for all $x \in G_i$

$$\tau_i(x) = \tau_j(i_{G_i,G_j}(x)) = \tau_j(r_{G_i,G_j}(x))$$

Then we say that **R** is **correctly typed** by (Σ, B). If all such typings are concrete, we say that **R** is **concretely correctly typed** by (Σ, B).

5 The SDLPTI Algorithm

SDLPTI stands for Simple Dataflow Linear Polymorphic Type Inference. Simple and linear because those are the only types of dataflow analysis and polymorphism respectively that we are concerned with. Actually we will see a little later that we can extend the applicability of the algorithm to certain non-linear cases as well.

From now on, we will restrict our attention to the simple imperative type discipline. Results for the crude type discipline are easily recovered by simply forcing the value of the partial function $\Sigma(S, \xi^{\mathrm{c}})$ to be independent of S, and then discarding S. In fact for the algorithm below all one need do is to delete the short underlined passages in step [10i] and the algorithm becomes correct for the crude type discipline. We assume that **S**, **Z**, **R** and (Σ, B) are fixed as before. We also insist that **S** and **R** are finite and that there are at least recursive algorithms for deciding membership of **Z** and (Σ, B). Finiteness of **S** and **R** is sufficient if theorem 3.2 is to serve as the basis for an iterative algorithm for determining $O_k(–)$ and $R(–)$ sets, and a finite number of $O_k(–)$ and $R(–)$ sets is sufficient to ensure termination of algorithm 5.3 below. We assume henceforth that a suitable collection of $O_k(–)$ and $R(–)$ sets have been given for the system **R** but we don't care whether they were obtained using the algorithm suggested by 3.2 or by some other magic.

Definition 5.1 Let Q be a set of t-symbols. Then $Q^\tau = \{\theta \mid S^\theta \in Q\}$. We say that Q and Q^τ are **acceptable** iff Q^τ contains at most one type constant. We say Q and Q^τ are **type singletons** iff Q^τ is a singleton.

of graph nodes are invariants of rewriting (particularly of redirection). The proof of this though, is no longer something that can be swept under the carpet as happens for TRSs. Given that we must irrevocably lose context freedom in the passage to the graph world, this is about the best that we could hope for.

The type theories we will construct are about the simplest that one could imagine under these circumstances. They resemble to some degree the phenomena found in conventional imperative languages like Pascal or Ada. The locality of the meaning of types is certainly reminiscent, but the type inference and parametric polymorphism aspects of our schemes are more general. The rest of this subsection sets up the general framework within which both of the type disciplines that we will develop fit. The following two subsections specialise this to the specific disciplines in question.

N.B. Due to pressure of space, and also for reasons of technical convenience, little motivatory material will occur among the definitions and theorems of this section and the next. On a first reading, readers may find it more convenient to briefly skim the definitions in the rest of this section and then skip to the discussion in section 6, where the salient properties of our type system are highlighted and compared to those of conventional systems. The informal impression gained thereby should help to make the structure and detailed content of the intervening technical material more accessible and digestible.

We assume we have an alphabet $\mathbf{Z}^{\mathrm{C}}$ of **type constants** and a disjoint alphabet $\mathbf{Z}^{\mathrm{V}}$ of **type variables**. $\mathbf{Z} = \mathbf{Z}^{\mathrm{C}} \cup \mathbf{Z}^{\mathrm{V}}$. We will use letters from the middle of the Greek alphabet as meta-variables standing for members of either alphabet. If we want to emphasise membership of either $\mathbf{Z}^{\mathrm{C}}$ or $\mathbf{Z}^{\mathrm{V}}$, we will superscript in the appropriate way, eg. ξ^{C} is in $\mathbf{Z}^{\mathrm{C}}$.

Definition 4.1 A **rule basis** Σ is a partial function $\Omega \times \mathbf{Z}^{\mathrm{C}} \to \mathbf{Z}^*$ where Ω is a set to be specified later. If $\Sigma(\omega, \xi^{\mathrm{C}})$ is defined (for $\omega \in \Omega$), then $((\omega, \xi^{\mathrm{C}}), \Sigma(\omega, \xi^{\mathrm{C}}))$ is called a **type rule** for ω and ξ^{C}. We write such a rule using the notation $\omega^{\xi^{\mathrm{C}} \leftarrow \xi_1 \ldots \xi_n}$, or $(\omega, \xi^{\mathrm{C}}) \leftarrow \xi_1 \ldots \xi_n$ if we wish to be less cluttered, where $\xi_1 \ldots \xi_n$ is the value of $\Sigma(\omega, \xi^{\mathrm{C}})$. A type rule for ω and ξ^{C} is **linear** iff $\Sigma(\omega, \xi^{\mathrm{C}})$ contains no more than one occurence of any $\xi^{\mathrm{V}} \in \mathbf{Z}^{\mathrm{V}}$. A rule basis is **linear** iff all its type rules are linear. The **arity** of a type rule, written $A(\omega, \xi^{\mathrm{C}})$ is the domain of $\Sigma(\omega, \xi^{\mathrm{C}})$, i.e. the set of indices of $\Sigma(\omega, \xi^{\mathrm{C}})$. No confusion will arise from this yet other arity concept.

Definition 4.2 An **axiom basis** B, is a partial function $\mathbf{S} \to \mathbf{Z}^{\mathrm{C}}$. When $B(S)$ is defined, if $\xi^{\mathrm{C}} = B(S)$, we say that $S^{\xi^{\mathrm{C}}}$ is a **typed symbol** (or **t-symbol**). More generally, we will also consider t-symbols S^{θ} for any $\theta \in \mathbf{Z}$.

Definition 4.3 A **type structure** (Σ, B) for a given set Ω fixed for the type discipline being considered, consists of a rule basis and an axiom basis. The structure is **linear** iff the rule basis is.

Definition 4.4 A **substitution** s is a map $\mathbf{Z} \to \mathbf{Z}$ which is the identity on $\mathbf{Z}^{\mathrm{C}}$, i.e. $s \in [\mathbf{Z}^{\mathrm{V}} \to \mathbf{Z}] \cup \{\mathrm{Id}_{\mathbf{Z}^{\mathrm{C}}}\}$.

4.1 A Crude Type Discipline

Let Ω be the one-point set $\{\bullet\}$. Then since $\Omega \times \mathbf{Z}^{\mathrm{C}} \cong \mathbf{Z}^{\mathrm{C}}$, it is preferable to drop all mention of Ω. We can therefore write a type rule as $(\xi^{\mathrm{C}}, \Sigma(\xi^{\mathrm{C}}))$ or as $\xi^{\mathrm{C}} \leftarrow \xi_1 \ldots \xi_n$ instead of using the more elaborate forms.

Now let some TGRS $\mathbf{R}$, and some type structure (Σ, B) be considered fixed, and let $\mathbf{S}$ and $\mathbf{Z}$ be the appropriate alphabets.

Definition 4.1.1 Let G be a graph and let $\tau : G \to \mathbf{Z}$ be a map from the nodes of G to type symbols. Then τ is a **typing** of G iff for all $x \in G$,

(1) If $B(\sigma(x))$ is defined, then $\tau(x) = B(\sigma(x))$.

(2) If $\Sigma(\tau(x))$ is defined, then $A(\tau(x)) = A(\sigma(x)) = A(x)$ and there is a(n x-dependent) substitution s_x such that for all $k \in A(x)$,

$T = \sigma(w)$. Let $S = \sigma(u)$ and $g'((p_k, c)) = (u_k, v)$ as before. Now $\sigma(v) \in O_l(\sigma(w)) = O_l(T)$ by hypothesis, and $\sigma(v) = \sigma(c)$ because g' is an extended matching. By (3), $O_l(T) = O_l(\sigma(w)) = O_l(\sigma(q)) \subseteq O_k(\sigma(p)) = O_k(\sigma(u)) = O_k(S)$, the penultimate step because g' is an extended matching. Thus $\sigma(v) = \sigma(c) \in O_k(\sigma(u))$ as required for (a). Since there are no redirections in the contractum building phase, $r_{G_i,G'_i} = i_{G_i,G'_i}$ so (b1) holds trivially.

Redirection Phase. Clearly there is nothing to prove for the non-redirected arcs in G_{i+1} since they are just injective copies under $i_{G'_i,G_{i+1}}$, of arcs of G'_i so (a) holds for them. Let $x \in G'_i$ be a redirected node, i.e. one whose incident arcs are to be redirected to $y \in G'_i$ because $(x, y) = (g'(a), g'(b))$ where (a, b) is a redirection of the rule governing the rewrite. By 2.2.(3) a is explicit. Let $S = \sigma(a)$ so $S = \sigma(x)$. Suppose b is explicit and $\sigma(b) = T$. Then by (2), $T \in R(S)$ and so $\sigma(y) \in R(\sigma(x))$. Hence $\sigma(r_{G'_i,G_{i+1}}(x)) \in R(\sigma(x))$ as required for (b1). Suppose alternatively b is implicit. Then there is an arc (q_l, b) in the left subpattern L of the rule such that (q_l, b) is an instance of $(T_l, -)$ say. Since g' is an extended matching, there is an arc $(t_l, y) \in G'_i$ with $g'((q_l, b)) = (t_l, y)$ so $\sigma(t) = T$ and thus by hypothesis $\sigma(y) \in O_l(T)$ and $\sigma(b) \in O_l(T)$. By (4), $O_l(T) \subseteq R(S)$ which gives $\sigma(y) \in R(\sigma(x))$ and $\sigma(r_{G'_i,G_{i+1}}(x)) \in R(\sigma(x))$ as before. Thus (b1) holds for an arbitrary redirection.

We need to show that (a) holds for the redirected arcs. Consider an arc $(w_m, x) \in G'_i$ where x is redirected. We know that $\sigma(x) \in O_m(\sigma(w))$. But also $\sigma(r_{G'_i,G_{i+1}}(x)) \in R(\sigma(x))$. Now $R(\sigma(x)) \subseteq O_m(\sigma(w))$ by (O_k). So $\sigma(r_{G'_i,G_{i+1}}(x)) \in O_m(\sigma(w))$ and (a) holds for the redirected arcs of G_{i+1}. ☺

Example 3.3 We have not presented a complete rule system in the pictures that illustrated rewriting and germ instances above, so strictly speaking we can't give an example of dataflow analysis. Nevertheless we can draw certain conclusions about the $O_k(-)$ and $R(-)$ sets of any system that includes the one rule that we did give. For instance, refering to the pattern P illustrated above, we must have Count′ $\in O_1$(Inc′), Plus $\in O_1$(Count′), O_1(Count) $\subseteq O_1$(Plus), 1 $\in O_1$(Plus), and also Inc′ $\in R$(Inc), Count′ $\in R$(Count). In addition, if rather than (root, r′) $\in$ *Red* we had had (root, a) $\in$ *Red*, then we would have had O_1(Count) $\subseteq R$(Inc) instead of Inc′ $\in R$(Inc). What other relationships hold between symbols, $O_k(-)$ sets and $R(-)$ sets depends of course on what other rules are present in the system.

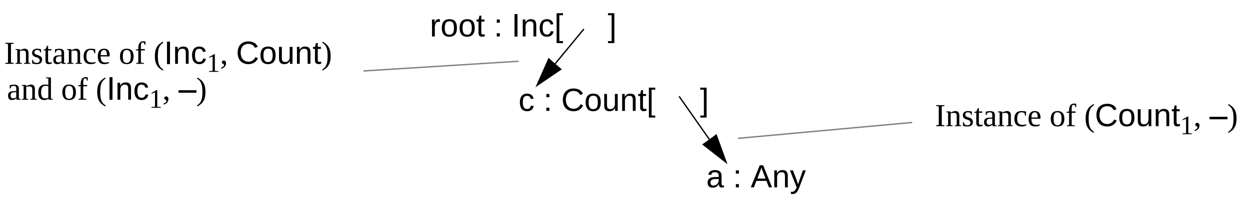
The above theorem can serve as a specification for an algorithm to determine suitable sets $R(-)$, $O_k(-)$, if the system is finite. The basic strategy is iteration until a least fixed point is reached using the conditions (R), (O_k), (1) – (4) as consistency conditions that drive various symbols into membership of the various $R(-)$ or $O_k(-)$ sets. See Banach (1992), Hankin (1991) for details. The analysis evidently provides a safe estimate of the sets of germ instances that actually occur during executions. In section 5, we will use the properties of the $R(-)$ and $O_k(-)$ sets as the basic driving engine for type inference once we have determined what types are to be in the present context.

4 Simple Type Theories For TGRSs

What should we mean by types for TGRSs? This is not a trivial question since type systems for conventional rewriting systems i.e. TRSs or the lambda calculus, are connected with issues such as strong normalisation. Since not even all TRSs (let alone TGRSs) have the strong normalisation property and since for those that do, the connection with type theories is built on precisely the cornerstone of structural induction over terms that we are forced to abandon, we must look for some weaker framework.

Since TGRSs are so general, it is not possible to demand the property that the type of an execution graph (whatever *that* might be) is an invariant of rewriting. This is the crucial (and suprisingly, seldom stated) subject reduction property of most type systems for TRSs that enables the strong normalisation results to go through. With its abandonment, the way is open to construct a more local concept of type, and for the particular notions of type that we will develop below, the invariant we will end up with, is that the types

An arc (p_l, c) of a graph G is an **instance** of a germ (S_k, T) iff $\sigma(p) = S$, $\sigma(c) = T$ and $l = k$. It is an instance of $(S_k, –)$ iff $\sigma(p) = S$ and $l = k$. These concepts may also be applied to arcs in patterns, provided we accept that arcs in which the child node is implicit may be instances only of implicit germs. For instance:



Let a fixed system be given. Our aim is to be able to estimate what germ instances actually occur during the rewriting of a system. Speaking loosely, we do this by estimating what symbols may occur at given child positions of any symbol and also what symbols a given symbol may get redirected to. This is done by an induction over the structure of executions.

Theorem 3.2 Assume a fixed system given. Suppose for all $S \in \mathbf{S}$, $k \in A(S)$ there are sets of symbols $R(S)$, $O_k(S)$ such that the following hold.

(R) $S \in R(S)$ and for all $T \in R(S)$, $R(T) \subseteq R(S)$.

(O_k) For all $T \in O_k(S)$, $R(T) \subseteq O_k(S)$.

Suppose also that for all rules $(incl : L \to P, root, Red)$ in the system we have (1) – (4) below.

(1) Let (p_k, c) be an instance of (S_k, T) in P where p and c are both explicit. Then $T \in O_k(S)$.

(2) Let (a, b) be in Red such that a and b are both explicit with $\sigma(a) = S$ and $\sigma(b) = T$. Then $T \in R(S)$.

(3) Let (p_k, c) be an instance of $(S_k, –)$ in P, and (q_l, c) be an instance of $(T_l, –)$ in L, where c is implicit and p is a contractum node. Then $O_l(T) \subseteq O_k(S)$.

(4) Let (a, b) be in Red such that b is implicit. Let (v_l, b) be an instance of $(T_l, –)$ in L and let $\sigma(a) = S$. Then $O_l(T) \subseteq R(S)$.

Then for every execution graph G_i, if $x \in G_i$ with $\sigma(x) = S$

(a) If (x_k, y) is an arc of G_i with $\sigma(y) = T$, then $T \in O_k(S)$,

(b) If G_j is some execution graph occurring later than G_i in the execution and $r_{G_i,G_j}(x) = z$ with $\sigma(z) = T$, then $T \in R(S)$.

Proof. The proof is by induction over the structure of executions. We will show that the contractum building and redirection phases of a rewrite, both preserve properties (a) and the one-step version of (b), (b1), i.e. where G_j is the successor of G_i in the execution; (b) then follows by induction and (R).

Base Case. The initial graph has no arcs so (a) holds trivially; and there is nothing to prove for (b1) since no rewrite created the initial graph.

Inductive Step. Suppose the hypotheses hold for $G_0 \ldots G_i$.

Contractum building Phase. Let G'_i be the graph after contractum building. Clearly (a) holds by hypothesis for $i_{G_i,G'_i}(G_i)$ so we need only consider the new arcs added during the building. Let (u_k, v) be such an arc and let it be the image under the extended matching $g' : P \to G'_i$ (where P is the pattern of the rule governing the rewrite) of an arc (p_k, c) in P. Now c is either explicit or implicit. If explicit, then (p_k, c) is an instance of some (S_k, T) and by (1), $T \in O_k(S)$. Since g' is an extended matching, $\sigma(u) = S$ and $\sigma(v) = T$ so $\sigma(v) \in O_k(\sigma(u))$. Alternatively, c is implicit. Since there must be a path of length at least one from the root of the left subpattern to c, c has an explicit parent in the left subpattern of the rule. Suppose (q_l, c) is the relevant arc. Since g' is an extended matching, $g'((q_l, c)) = (w_l, v)$ where $w \in i_{G_i,G'_i}(g(L))$. Let

Definition 2.8 An **initial graph** is one consisting of an isolated node with empty arity, labelled by the symbol Initial.

Definition 2.9 A **system** is a set of rules $\mathbf{R}$. An **execution** of $\mathbf{R}$ is a sequence of graphs $[G_0, G_1 \ldots]$ of maximal length such that

(1) G_0 is initial,

(2) For all $i \geq 0$ such that i+l is an index, there is a rule $D \in \mathbf{R}$ and $x \in G_i$ such that G_i is the pre-graph and G_{i+1} the post-graph of a rewrite of G_i at x according to D.

Note that the above definition does not address garbage collection or reduction strategy, let alone fairness or other issues of concern in concurrency theory[1]. TGRSs may be enhanced with notions that would shed light on these things (see eg. Glauert et al. (1988a, b), Banach (1991c, 1992)) but to do so here would clutter the exposition needlessly.

Definition 2.10 Any graph that occurs in an execution of some system is an **execution graph** of that system.

Definition 2.11 Let G_i be an execution graph and G_{i+1} be its successor in an execution. Let G'_i be the corresponding graph after contractum building. The functions i_{G_i,G'_i}, r_{G_i,G'_i}, $i_{G'_i,G_{i+1}}$, $r_{G'_i,G_{i+1}}$ are defined as follows:

(1) i_{G_i,G'_i} is the natural injection of nodes of G_i to nodes of G'_i.

(2) $r_{G_i,G'_i} = i_{G_i,G'_i}$.

(3) $i_{G'_i,G_{i+1}}$ is the natural bijection of nodes of G'_i to nodes of G_{i+1}.

(4) $r_{G'_i,G_{i+1}}(x) = i_{G'_i,G_{i+1}}(x)$ unless x has been redirected, i.e. it has had its incident arcs swung over to point to some other node $y \neq i_{G'_i,G_{i+1}}(x) \in G_{i+1}$ during the redirection phase, in which case $r_{G'_i,G_{i+1}}(x) = y$.

We define $i_{G_i,G_{i+1}}$ as the composition $i_{G'_i,G_{i+1}} \circ i_{G_i,G'_i}$, and similarly for $r_{G_i,G_{i+1}}$, $i_{G_i,G_{i+k}}$, $r_{G_i,G'_{i+k}}$ etc. Thus $i_{G_i,G_{i+n}}(x)$ is the copy of x in G_{i+n}, while $r_{G_i,G_{i+k}}(x)$ follows the redirection history of x and is the node of G_{i+n} that copies of x have been redirected to.

This completes the description of the standard operational semantics of general term graph rewriting. It turns out that the model has an elegant universal reformulation, but it would take us too far out of our way to go into the details of this. See Banach (1991a, b) for a fuller description.

3 Simple Dataflow Analysis

We shift our attention from execution graphs as such, to relations between symbols that estimate the possible local structures occurring in them. In particular we are concerned with the occurrences of symbols at parent and child ends of an arc of an execution graph and the redirection histories of nodes.

Definition 3.1 If $S, T \in \mathbf{S}$ and $k \in A(S)$, then the object

$$\Gamma = (S_k, T)$$

is called an **explicit** k**-germ**. An object

$$\Gamma' = (S_k, -)$$

is called an **implicit** k**-germ**.

1. TGRSs were initially invented with the objective of providing a model for graph reduction implementations of functional languages, parallel as well as serial.

Definition 2.6 (Redirection) Let $D = (incl : L \to P, root, Red)$ be a rule with left subpattern L, G a graph, g a matching of L to G and G' the graph resulting from the construction of 2.4. We construct the graph H given by

(1) $N_H = N_{G'}$,

(2) $\sigma_H = \sigma_{G'}$,

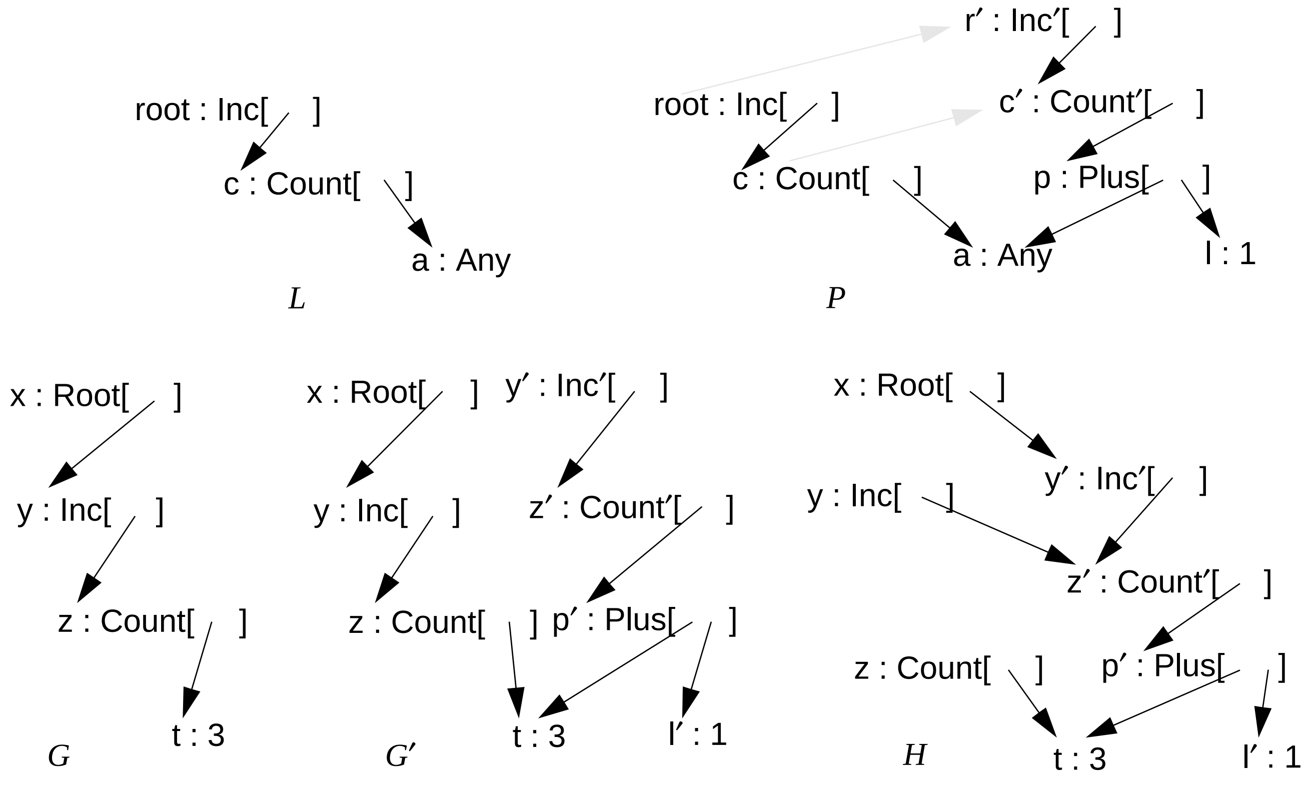
(3) $$\alpha_H(\{(1,x)\})[k] = \begin{cases} \{(2,y)\ldots\} & \text{if } (u,y) \in Red \text{ for some } y \in P \text{ and } u \in g'^{-1}(\alpha_{G'}(\{(1,x)\})[k]) \\ \alpha_{G'}(\{(1,x)\})[k] & \text{otherwise} \end{cases}$$

$$\alpha_H(\{(2,n)\})[k] = \begin{cases} \{(2,y)\ldots\} & \text{if } (u,y) \in Red \text{ for some } y \in P \text{ and } u \in g'^{-1}(\alpha_{G'}(\{(2,n)\})[k]) \\ \alpha_{G'}(\{(2,n)\})[k] & \text{otherwise} \end{cases}$$

$$\alpha_H(\{(1,x),(2,n_1)\ \ldots\ (2,n_m)\})[k] =$$
$$\begin{cases} \{(2,y)\ldots\} & \text{if } (u,y) \in Red \text{ for some } y \in P \text{ and } u \in g'^{-1}(\alpha_{G'}(\{(1,x)\ldots\})[k]) \\ \alpha_{G'}(\{(1,x),(2,n_1)\ \ldots\ (2,n_m)\})[k] & \text{otherwise} \end{cases}$$

Definition 2.7 If D, G and g are as in 2.4, 2.5, then a **rewrite** of G at x according to D is defined to yield the graph H. G is said to be the **pre-graph**, and H the **post-graph** of the rewrite.

In plain language, the graph G' glues copies of the contractum nodes of P onto G, ensuring that arcs are introduced in such a way that the extended matching $g' : P \to G'$ exists. Another way of visualising the same thing is to take disjoint copies of G and P, and for all redex nodes x, to "pinch together" x and its primage under g. This is similar to the formal construction. Likewise, the redirection phase locates the images in G' of the redirection pairs of D, and swings all arcs incident on the image of the LHS to point to the image of the RHS. There is no ambiguity about redirection as the LHS's of distinct redirections are labelled with different symbols so their images under matching cannot coincide. The diagrams below provide a small example of rewriting. The objects occuring there are named after their role in the above theory. Note that the faint dotted arrows represent the redirections of the rule.



Nodes x with $\sigma(x) = \mathsf{Any}$ are called **implicit**, whereas others are **explicit**. More generally we allow ourselves to regard a graph as a pattern for which the number of implicit nodes is zero when convenient, but we never normally regard a pattern as a graph. Patterns and graphs are not normally deemed to have roots unless we specifically say so or it is clear from context.

Definition 2.2 A **rule** D is a triple $(P, root, Red)$ where

(1) P is a pattern known as the pattern of the rule.

(2) $root$ is an explicit node of P called the **root** and all implicit nodes are accessible from $root$. Also if $\sigma(root) = S$, then D is a **rule for** S.

(3) Red is a set of pairs (called **redirections**) of nodes of P such that if $(x, y) \in Red$, then x is explicit, and accessible from $root$. Also if (x, y), $(u, v) \in Red$, then if $x = u$ then $y = v$, and if $x \neq u$, then $\sigma(x) \neq \sigma(u)$. For $(x, y) \in Red$, x is called the **LHS** of the redirection while y is the **RHS**.

The subpattern of P accessible from (and including) the root is called the **left subpattern** of the rule and is usually denoted by L, while nodes of P not in the left subpattern are called **contractum** nodes. Thus another, slightly redundant way of writing a rule is $D = (incl : L \rightarrow P, root, Red)$ where $incl$ is the inclusion of the left subpattern into P.

Definition 2.3 A **matching** or homomorphism of a pattern P with root $root$ say, to a graph (or pattern) G at a node $t \in G$ is a map $h : P \rightarrow G$ such that

(1) $h(root) = t$

(2) If $x \in P$ is explicit, then $\sigma(x) = \sigma(h(x))$, $A(x) = A(h(x))$, and for all $k \in A(x)$, $h(\alpha(x)[k]) = \alpha(h(x))[k]$.

Thus a matching is a type of graph homomphism in which implicit nodes may match anything but explicit nodes have to behave well. By dropping the condition (1) involving roots, this definition will also suffice for matching general patterns to graphs or other patterns, or for matching graphs to other graphs.

We now turn to the rewriting model itself. An informal summary and diagramatic example follow the formal definitions below.

Definition 2.4 (Contractum Building) Let $D = (incl : L \rightarrow P, root, Red)$ be a rule with left subpattern L and let G be a graph. Let $g : L \rightarrow G$ be a matching of L to G at some node of G. Then $g(L)$ is called the **redex**. We build first the graph G' given by

(1) $N_{G'} = (N_G \uplus N_P)/{\approx}$ which is the disjoint union of N_G and N_P factored by the equivalence relation $\approx$, where $\approx$ is the smallest equivalence relation such that[1] $(1, x) \approx (2, n)$ whenever $g(n) = x$.

(2) $\sigma_{G'}(\{(1, x)\}) = \sigma_G(x)$,
$\sigma_{G'}(\{(2, n)\}) = \sigma_P(n)$,
$\sigma_{G'}(\{(1, x), (2, n_1) \ldots (2, n_m)\}) = \sigma_G(x)$,
which is consistent because g is a matching.

(3) $\alpha_{G'}(\{(1, x)\})[k] = \{(1, \alpha_G(x)[k]) \ldots\}$ for $k \in A(x)$,
$\alpha_{G'}(\{(2, n)\})[k] = \{(2, \alpha_P(n)[k]) \ldots\}$ for $k \in A(n)$,
$\alpha_{G'}(\{(1, x), (2, n_1) \ldots (2, n_m)\})[k] = \{(1, \alpha_G(x)[k]) \ldots\}$ for $k \in A(x)$,
which is again consistent since g is a matching. The … on the RHS of these cases indicates that the k'th child of say a singleton equivalence class node of G', eg. $\{(1, x)\}$, may not itself be a singleton.

Lemma 2.5 There is a matching $g' : P \rightarrow G'$ that extends $g : L \rightarrow G$ (allowing for obvious identification of nodes modulo disjoint union etc.).

Proof. Define $g'(2, n) = \{(2, n) \ldots\}$. It is clear that g' has the appropriate properties. ☺

1. The notation $(1, x)$ or $(2, n)$ tags each element of the binary disjoint union with the tag 1 or 2 to unambiguously indicate its origin. Likewise $\{(1, x) \ldots\}$ denotes the equivalence class containing $(1, x)$.

Given these shortcomings of the graph world, what weapons can we use to overcome the lack of structural induction and the complexity of matchings? The crude answer is that we must resort to induction over the structure of executions instead of induction over the structure of graphs (since the latter doesn't exist in any sense useful to us), and having that, to dataflow analysis of the rule system (the validity of which is itself established by the former technique). In fact, the former technique is not so different from what happens in the term world (where it leads to the subject reduction property), but in the term world, the triviality of several of the steps is such that they are passed over without mention, and the only issues of interest that remain can be dealt with by induction over terms. This is also related to the context freedom mentioned previously. A further issue related to this is rulewise modularity. Many term based systems have the property that certain semantic attributes of a rule can be considered in isolation from those of the rest of the rules in the system, again reducing induction over executions to induction over terms. This is a pleasing and desirable feature. Unfortunately context freedom and rulewise modularity are the main casualties of the passage to the graph world. There must be *some* casualties of course, we cannot expect significant generalization without paying some price. The lesson for the graph world though, boils down to the fact that we can no longer consider subsystems in isolation from one another as easily as we can in the term world. The entire system must in principle be taken into account when considering even some seemingly local properties of some small part of it.

The structure of the rest of the paper is as follows. In Section 2 we define our TGRSs as a generalisation of the model of Barendregt et al. (1987). The terminology is a little non-standard for convenience. Section 3 deals with simple dataflow analysis, showing how the structure of a rule system permits information about dynamic occurrences of so-called germ instances in execution graphs to be infered statically. Section 4 considers the question of how types are to be defined for TGRSs, and comes up with a simple scheme that is refined into two specific type disciplines in subsections 4.1 and 4.2. These are called the crude type discipline and the simple inperative type discipline respectively. Section 5 shows the results of section 3 can be used to show the soundness of a unification-based type inference algorithm for the type disciplines introduced in section 4. Section 6 contains a discussion, suggests extensions, and concludes.

2 Term Graph Rewriting

We assume we are given an alphabet $\mathbf{S} = \{S, T\ldots\}$ of **node symbols**. We write $\{1\ldots n\}$ for the set of naturals between 1 and n inclusive. We let $\{1\ldots 0\} = \varnothing$ and let $\mathbf{SeqN}$ be the set of all such subsets of $\mathbf{N}$, including $\varnothing$.

Definition 2.1 A **term graph** (or just **graph**) G is a triple (N, σ, α) where

(1) N is a set of nodes,

(2) σ is a map with signature $N \to \mathbf{S}$,

(3) α is a map with signature $N \to N^*$.

Thus σ maps a node to the node symbol that labels it, and α maps each node to its sequence of successors. We write $A(x)$, the **arity** of a node, for the domain of $\alpha(x)$, i.e. the member of $\mathbf{SeqN}$ consisting of indices of $\alpha(x)$. We also write $x \in G$ instead of $x \in N(G)$ etc. Lastly we subscript N, σ, α with the name of the graph in question whenever more than one graph is being discussed. A successor of a node determines an arc of the graph, and we will write such an arc as (p_k, c), to indicate that the child c is the k'th child of the parent p, i.e. that $c = \alpha(p)[k]$ for some $k \in A(p)$. The names are intended to be somewhat alliterative: N for nodes, σ for symbols, α for arcs.

We will assume for the remainder of this paper that symbols have fixed arities, i.e. that there is a map $A : \mathbf{S} \to \mathbf{SeqN}$ such that if $\sigma(x) = S$, then $A(x) = A(S)$ (the different A's should not cause confusion).

We will also assume there is a special node symbol Any, not normally considered to be in $\mathbf{S}$, and if any nodes of a graph are labeled with Any, we call such a graph a **pattern**. We further assume that

$$\sigma(x) = \mathsf{Any} \;\Rightarrow\; \alpha(x) = \varepsilon \qquad (\varepsilon = \text{the empty sequence}).$$

SIMPLE TYPE INFERENCE FOR TERM GRAPH REWRITING SYSTEMS

R. Banach[1]

Computer Science Department, Manchester University,

Manchester, M13 9PL, U.K.

Abstract

A methodology for polymorphic type inference for general term graph rewriting systems is presented. This requires modified notions of type and of type inference due to the absence of structural induction over graphs. Induction over terms is replaced by dataflow analysis.

1 Introduction

Term graphs are objects that locally look like terms, but globally have a general directed graph structure. Since their introduction in Barendregt *et al.* (1987), they have served the purpose of defining a rigorous framework for graph reduction implementations of functional languages (Peyton-Jones (1987)). This was the original intention. However the rewriting of term graphs defined in the operational semantics of the model, makes term graph rewriting systems (TGRSs) interesting models of computation in their own right. One can thus study all sorts of issues in the specific TGRS context. Typically one might be interested in how close TGRSs are to TRSs and this problem is examined in Barendregt et al. (1987), Farmer et al. (1990), or Kennaway et al. (1991).

In this paper we examine a related issue, that of type inference. There are two ways that we could approach this question. The first is to look for conditions on TGRSs that ensure we recover the results that exist for TRSs of various kinds. This has the virtue of finding the best generalizations of known TRS results in the graph world but leaves the question of what type inference for general TGRSs might look like, unanswered. The second approach addresses the latter question head on, and attempts to construct type theories for TGRSs directly, without close reference to the TRS results. This is the approach we will follow.

Two main obstacles have to be overcome when we consider the graph world as opposed to the term world. These are the collapse of structural induction, and the triviality of term matching compared with graph matching. For terms, structural induction is the mainstay of most proofs of significant results, and it is hard to see what to replace it with. A byproduct of structural induction is the fact that many results are context free in the sense that if Π is some property that holds for a set of terms T, then if $C[\ ]$ is some context, $C[t]$ satisfies Π for all $t \in T$ and contexts $C[\ ]$. Such results frequently hold in TRSs related to the lambda calculus (see Barendregt (1984), Girard et al. (1989), Huet (1990)).

For terms, we also have that if a rule matches some subterm, then apart from variable instantiations, the subterm is an exact copy of the rule LHS. This leads to a number of subtle properties that terms have compared with the more general term graphs, and is founded on the fact that terms are simply trees. In the more general graph world, we have to be more careful about the matching problem as matchings are not just isomorphisms with variable instantiation, and we have to confront the fact that important properties of a given rule are by no means required to hold for some arbitrary execution graph that its LHS may match.

1. Email: `rbanach@cs.man.ac.uk`