Graph Types*

Nils Klarlund & Michael I. Schwartzbach
{klarlund,mis}@daimi.aau.dk

Aarhus University, Department of Computer Science,
Ny Munkegade, DK-8000 Århus, Denmark

Abstract

Recursive data structures are abstractions of simple records and pointers. They impose a *shape* invariant, which is verified at compile-time and exploited to automatically generate code for building, copying, comparing, and traversing values without loss of efficiency. However, such values are always tree shaped, which is a major obstacle to practical use.

We propose a notion of *graph types*, which allow common shapes, such as doubly-linked lists or threaded trees, to be expressed concisely and efficiently. We define regular languages of *routing expressions* to specify relative addresses of extra pointers in a canonical spanning tree. An efficient algorithm for computing such addresses is developed. We employ a second-order monadic logic to decide well-formedness of graph type specifications. This logic can also be used for automated reasoning about pointer structures.

1 Introduction

Recursive data types are abstractions of structures built from simple *records and pointers*. The values of a recursive data type form a set of pointer structures that all obey a common *shape invariant*. The advantage of this approach is twofold:

- validity of the invariant can be statically verified at compile-time, which contributes to the correctness of programs; and
- the invariant can be exploited to automatically generate code for such tasks as copying, comparing, and traversing values.

Recursive data types originate from the seventies [7] and have become ubiquitous in modern typed functional languages such as ML [8] and MIRANDA [10], but they may also be employed in PASCAL-like imperative languages. Their benefits are substantial, but they also impose limitations; in particular, the values of recursive data types will always be *tree shaped*. In this paper we present a natural generalization, *graph types*, which allows a large variety of *graph shaped* values, including (doubly-chained) cyclic lists, leaf-to-root-linked trees, leaf-linked trees, and threaded trees.

The key idea is to allow only graphs with a *backbone*, which is a canonical spanning tree. All extra edges must depend functionally on this backbone. The extra edges are specified by a language of regular *routing expressions*, which give relative addresses within the backbone. We show that construction of such graph values—along with all relevant manipulations—can happen efficiently in linear time. We introduce a decidable *monadic logic* of graph types, which allows automatic

*This paper will also be presented at POPL'93; references should cite the proceedings.

derivation of some constant time operations—such as concatenation of doubly-linked lists. There have been other attempts to describe graph-shaped values. Our proposal, however, allows exact descriptions of a more general class of types, and it does so using an intuitive notation that is very close to existing concepts in programming languages.

This summary is kept in an informal, explanatory style. Formal definitions and algorithms are included in the appendix.

2 Data Types

For this presentation, a (recursive) data type $\mathcal{D}$ is a special kind of tree grammar. The nonterminals are called *types*. There is a distinguished *main* type, which in examples is always the one mentioned first; the others are merely auxiliary. A *production*

$$T \to v(a_1 : T_1, \ldots, a_n : T_n)$$

of $\mathcal{D}$, where T and the T_i's are types, declares a *variant* v of type T containing *data fields* named $a_1, \ldots, a_n$; we say that the production declares a *type-variant* $(T{:}v)$. For each type, the possible variants must be mutually distinct; thus $(T : v)$ uniquely determines the production. Moreover, for each type-variant, the data fields must be mutually distinct.

The values of a data type are essentially the derivation trees of the underlying context-free grammar, starting with the main type. They are implemented as pointer trees, but the programmer will never directly manipulate these pointers. Each node of such a pointer tree is an instance of a variant of a type. A formal definition of the values of a data type is given in section A1 of the appendix. As a simple example, consider the following data type, which specifies a type of simple integer lists

L → nonempty(head: Int, tail: L)
  → empty()

We can think of the type Int as being a data type specified as

Int → 0() | 1() | 2() | ...

We allow *implicit* variants as a form of syntactic sugar. If the sets of data fields are distinct for all variants, then the explicit variants are not needed; we may think of the variant names as being a concatenation of the field names. Thus, we may instead write

L → (head: Int, tail: L)
  → ()

Programming with Data Types

When a data type has been specified, it gives rise to a number of operations in the programming language. First of all, there is a language for denoting *constant* values. For the above lists, one may write down

L(head:11,tail:(head:12,tail:(head:13,tail:())))

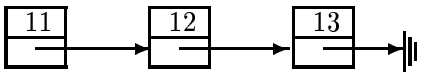
for the list of type L with elements 11, 12, and 13. If x is a variable containing a value of type L, then x.tail.tail.head specifies the *address* of a subtree, in this case of type Int. In a functional language this would always denote the corresponding value; in an imperative language there is the usual distinction between *l*- and *r*-values. The *comparison* x = y is always defined for two values of type L. If x is a value of type L, then the boolean expression is(x,v) yields true exactly when x is of variant v. In an imperative language, the *value assignment* x := y is present, possibly accompanied by the *swap* x :=: y which exchanges two subtrees without copying. Values of data types are traversed by recursive functions or procedures. Thus, explicit pointers are never used.

There is no intrinsic loss of efficiency in this approach. Constants can be built, copied, compared, and traversed in optimal linear time, and addresses are accessed in constant time. Thus, if one really wants tree-shaped values, then only advantages are to be seen.

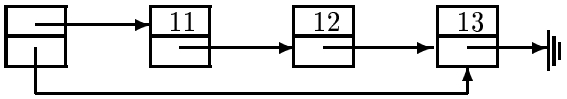
Shortcomings of Data Types

The main draw-back of data types is the limited shapes of values that they allow. For

the above simple lists, values always look as follows (an empty record is pictured as a "ground" symbol)



However, it is a common optimization to want an extra pointer to gain constant time access to the last element of the list. Thus, the values should instead have the following shape



These are not trees and, hence, cannot be specified by data types. Until now, there has been no solution to this problem. The only possibility has been to revert to the often perilous use of explicit pointers.

3 Graph Types

We introduce the notion of *graph types*, which form a conceptually simple extension of data types. They allow graph shaped values while retaining the efficiency and ease of use. There are two key insights to our solution:

- while being graphs, the values all have a *backbone*, which is a canonical spanning tree; and
- the remaining edges are all functionally determined by this backbone.

Many, but not all, sets of graphs fit this mold; we give examples of both kinds.

A *graph type* extends a data type by having *routing fields* as well as *data fields*. Productions now look like

$$T \to v(\ldots a_i : T_i \ldots a_j : T_j[R] \ldots)$$

Here a_i is a normal data field but a_j is a routing field. It is distinguished by having an associated *routing expression* R. A graph type has an underlying data type, which is obtained by removing the routing fields. The backbones of the graph type values are simply the values of this data type. Routing expressions describe relative addresses within the backbone. The complete graph type value is obtained by using the routing expressions to evaluate the destinations of the routing fields.

Routing expressions are regular expressions over a language of *directives*, which describe navigation within a backbone. Directives include "move up to the parent (from a specific child)" ($\uparrow$ or $\uparrow a$) , "move down to a specific child" ($\downarrow a$), and "verify a property of the current node", where properties include "this is the root" ($\wedge$), "this is a leaf" (\$), and "this is (a specific variant of) a specific type" (T or $(T : v)$). A routing expression defines the destination indicated by the corresponding routing field if its regular language contains precisely one sequence of successful directives leading to a node in the tree. A graph type is *well-formed* if every routing expression always defines a unique destination. Section A2 of the appendix gives formal definitions of these concepts.

To make a convincing case for this new mechanism, we need to demonstrate the following facts:

- many useful families of structures can be easily specified;
- values can be manipulated at run-time similarly to values of data types, and without loss of efficiency; and
- well-formedness of graph type specifications can be decided at compile-time.

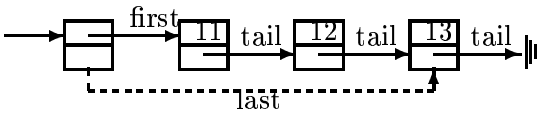
4 Examples

We now show that many common pointer structures have simple specifications as graph types. The examples are all well-formed, which can be easily seen in each case. In pictures of values, we use the convention that pointers from data fields are solid, whereas

those from routing fields are dashed. The root of the underlying spanning tree, or backbone, is indicated by a solid pointer with no origin. The list with a pointer to the last element looks like

H → (first: L, last: L[↓first ↓tail* $ ↑])
L → (head: Int, tail: L)
→ ()

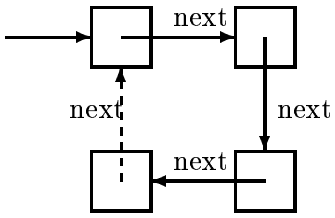
A typical value is



The routing expression ↓first ↓tail* $ ↑ for the "last" field contains the following directives: move down along the "first" pointer (↓first); follow the "tail" pointers until a leaf is reached (↓tail* $); then back up once (↑). This is the destination of the "last" pointer. A cyclic list looks like

C → (next: C)
→ (next: C[↑* ∧])

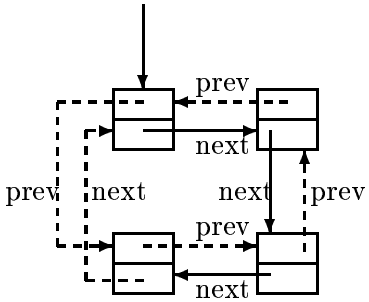
A typical value is



The routing expressions contain the following simple directives: move up to the root. A doubly-linked cyclic list looks like

D → (next: D, prev: D[↑ + ∧ ↓next*$])
→ (next: D[↑* ∧], prev: D[↑ + ∧])

A typical value is

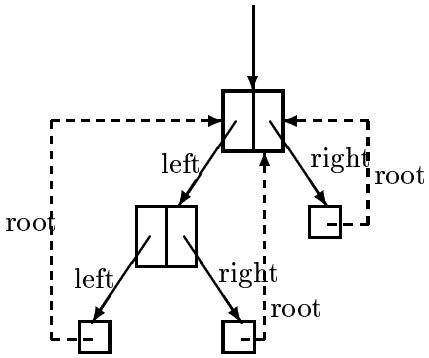


Directives are more complicated here; they use the nondeterministic union operator on regular expressions (+) to express context-dependent choices. For example, consider the "prev" field of the first variant. According to the routing expression ↑ + ∧ ↓next*$ of this field, we must either move up, or, if we are at the root, follow "next" pointers to the leaf.

A binary tree in which all leaves are linked to the root looks like

R →(left, right: R)
→(root: R[↑* ∧])

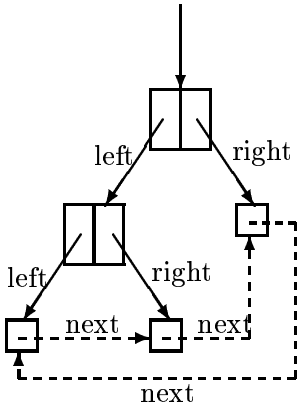
A typical value is



A binary tree in which all the leaves are joined in a cyclic list looks like

J →(left, right: J)
→(next: J[STEP$])

where STEP abbreviates ↑right*(↑left↓right+ ∧) ↓left*. A typical value is



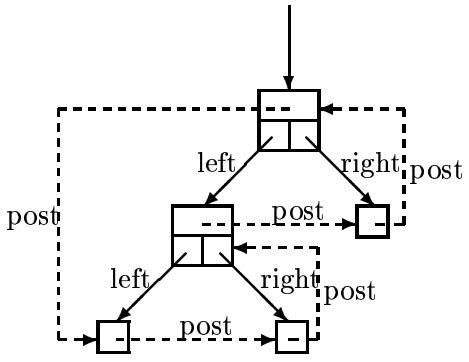
A binary tree with *red* or *black* leaves, in which those of the same color are joined in a cyclic list, looks like

K →(left, right: K)
→red(next: K[BLACK* RED])
→black(next: K[RED* BLACK])

where RED abbreviates STEP (K:red) and BLACK abbreviates STEP (K:black). We shall abstain from showing a typical value of this type. Finally, a binary tree in which all nodes are threaded cyclically in post-order looks like

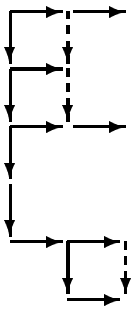
T →(left, right: T, post: T[POST])
→(post: T[POST])

where POST abbreviates ↑right+↑left↓right↓left*$+ ∧ ↓left*$. A typical value is



At a first glance such specifications may seem daunting, but at least to the authors they quickly became familiar. The use of abbreviations, such as STEP and POST above, may improve legibility and promote reuse of routing expressions. Complicated pointer structures may give rise to complicated graph type specifications. However, it is fair to say that the complexity of the graph type specification correlates well with this inherent complexity, in the same way that a verbal or pictorial description would.

Not all families of graph shaped values can by specified by graph types. First of all, they must be *deterministic*, in the sense that all edges must be *functions* of some underlying spanning tree. This precludes such things as a pointer from the root to *some* node in the tree. But even all deterministic situations cannot be specified. Consider a generalized *tableau* structure on a grid, in which there must be an edge from a point to the one immediately below, if they are both present.



A graph type cannot represent such graphs, since the variant at a given node is dependent on whether there is a downward pointing edge. Thus the variant is dependent on the rest of the graph—something we cannot specify in a context-free grammar.

5 Programming

So far, we have seen that many families of pointer structures can be captured as the values of graph types. We must also demonstrate that they can be used for programming in a manner similar to that for data types.

An obvious problem with having graph shaped values is that the recursive traversal may be problematic; how can we avoid cycles? However, for graph types we have the

canonical spanning tree of the underlying data value. Thus, many of the simple techniques can be inherited in a straightforward manner. For example, the algorithm for comparing two graph values is exactly the same as for the underlying two data values; the routing fields are just ignored.

The syntax for constants are also the same as for the underlying data type. The values of the routing fields are then computed automatically. The example values of the previous section are specified as constants as follows:

```
H(first: (head: 11, tail: (head: 12,
         tail: (head: 13, tail: ()))))
C(next: (next: (next: ())))
D(next: (next: (next: ())))
R(left: (left: (), right: ()), right: ())
J(left: (left: (), right: ()), right: ())
T(left: (left: (), right: ()), right: ())
```

Note that the expressions for the C- and D-values are identical, as are those for the R-, J-, and T-values.

Copying (sub)values happens in two steps. First, the underlying spanning tree is copied; second, the values of the routing fields must be reevaluated. Consider for example the leaf-to-root-linked tree. If a subtree is copied, then the leaves must now point to the new root of that tree.

If a data field in a graph value is assigned, then several routing fields in the both the surrounding spanning tree and the new graft may have to change. Consider for example the red-black leaf-linked trees. If a leaf is changed from red to black, then it must be removed from one cyclic list and inserted in another. A simple way of handling this is to reevaluate all routing fields, but that is undesirable since the surrounding tree may be large and the graft may be small. A similar problem exists for the swapping of subtrees. We must develop an algorithm for detecting the routing fields that are required to be updated.

Routing fields can be read just like data fields; they also point to subtrees of the canonical spanning tree. It is, of course, not possible to assign directly to a routing field.

In summary, many of the required algorithms are inherited from the underlying data structure. However, we must be able to evaluate *all* routing fields in only combined linear time, and for assignment we need to detect those routing fields that must be updated.

Evaluating Routing Fields

Backbones can clearly be constructed in linear time. Given a backbone, it is possible to evaluate all routing fields in combined linear time.

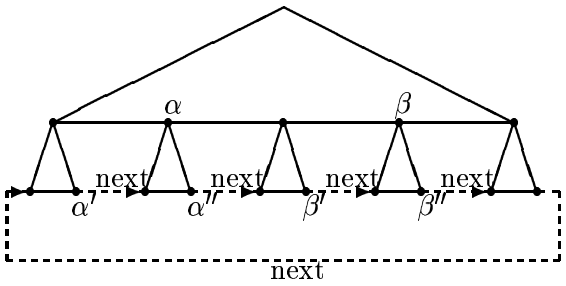
First, each routing expression in the graph type is translated into an equivalent nondeterministic automaton. This translation is linear.

Next, a table is constructed that for each node α and for each automaton state q of each automaton A contains a pointer. Intuitively, if this pointer is not nil, it indicates a node β reachable by a sequence w of directives from α such that upon reading w, automaton A may end up in a final state at node β. This table is calculated in linear time by an algorithm described in the appendix.

When the table has been constructed, the destination of a routing field at α is given as the pointer found in an entry (α, q^0) of the table, where q^0 is an initial state of the automaton representing the routing expression.

Detecting Required Updates

Sometimes when a change occurs, it is sufficient to update routing fields for only a small part of the value. For example, this happens when swapping subtrees of values of type J, the type of leaf-linked binary trees. Consider the situation after the subtrees rooted at addresses α and β have been swapped:



Here only the "next" pointers at α', α'', β', and β'' need to be updated. If we assume that J is made doubly-linked—by adding a field "prev: J[↑ + ∧]"— it would often be less costly to locate the four nodes $\{\alpha', \alpha'', \beta', \beta''\}$ after the change and reevaluate their "next" fields than evaluating all routing expressions in the backbone from scratch. In fact, with this approach we can guarantee that the time to locate fields in need of updating is proportional to the total length of the paths that lead to these fields, in this case of the paths from α to α', from α to α'', from β to β', and from β to β''.

To generate these paths, we consider each node incident on a backbone edge that changes (above, it would be α, β, and their parents). Each automaton state at such a node can be followed backwards—towards possible *origins*, routing fields whose routes go through the node—and forwards—towards a possible destination. Above, this involves finding four destinations and four origins. For example, when considering α, we obtain two origins, the "next" fields of α' and α'', and their corresponding destinations.

We shall shortly see how further optimizations are possible. Note, however, that for some graph types the number of paths to follow may be proportional to n. This happens for example for the root linked trees of type R described earlier when a new root is added to an existing tree. In this case there is no gain in using the techniques described in this section compared to the algorithm for updating all routing fields.

Monadic Logic and Well-Formedness

The monadic second-order logic on graph types is a logical formalism that allows several important properties about graph types to be expressed. In section A4 of the appendix, we define the logic formally and show that it is decidable. Our logic permits quantification over values of graph types, addresses, and sets of addresses. In this logic we can formulate questions such as "What is the type-variant of a node α in a value x?" or "Is there a walk in a value x from node α to node β according to a routing expression R?" The question of whether a graph type is well-formed can also be expressed in the logic as it is shown in section A4 of the appendix. Thus this question is decidable. Similarly, questions about comparing values, such as $\mathbf{Val}\,\mathcal{G}_1 \subseteq \mathbf{Val}\,\mathcal{G}_2$, where $\mathcal{G}_1$ and $\mathcal{G}_2$ are graph types, are decidable.

Although much can be expressed in the monadic second-order logic on graph types, there are simple operations that cannot. For example, one cannot represent the result of replacing a subtree with another subtree (although certain properties of the result may be expressible).

Access Optimizations

In the example of updating routing fields in leaf-linked trees, we saw that only four fields needed to be updated. It is not hard to see that calculating the destination of each such routing field is not necessary. For example, the new value of the "next" field at α' is the old value of the "next" field at β'. Thus, when the four routing fields have been located, the updates can take place in constant time by properly permuting the values of known "next" pointers. Such use of the values of routing fields is called *access optimization*.

The formal reasoning behind access optimization can be formulated in monadic logic. For example the question "Is the value of the "next" field at α' in the new graph the same

as the value of the "next" field at β' in the old graph?" can be expressed, and the answer "yes" can be computed.

In general, a strategy for access optimization is to compare values contained in nodes already located to the destination of paths that arise in the detection of required updates. This involves trying out different combinations of paths that are followed explicitly and testing whether other needed destinations or origins can be found in constant time. Thus one can formulate a minimization problem for finding the least number of paths that need to be followed in order to carry out an update, and this problem is decidable.

For doubly-linked lists of type D, such reasoning allows the automatic generation of optimal, constant-time code for concatenating lists—without the programmer having to specify any pointer operations.

6 Related Work

Decidability of logics of graphs have been studied extensively; see [4] for references to the classical results that the monadic second order logic on finite trees is decidable and for extensions to more general graphs. The hyperedge-replacement grammars of [4] and similar context-free graph rewriting formalisms describe much larger classes of graphs than our graph types. An important result of [4] is that any property expressed in second-order monadic logic on graphs is decidable on hyperedge-replacement grammars. We could have used this result to derive our decidability result; but the translation into context-free graph grammars appears to be more complex than our approach. Although mathematically interesting, context-free graph grammars tend to be hard to understand; this is likely the reason why, to our knowledge, they have not been used for describing types in programming languages.

Closer in spirit to our approach are the *feature grammars and algebras*; see [5] for references. These formalisms are built on the view that features (corresponding to our record fields) are partial functions that identify attributes. Not being based on tree structures, features allow the description of self-referential data structures. As opposed to our approach, the values designated are not guided by any expressions.

The programming languages in [1, 2] and [3] use similar ideas and permits circular data structures. A restriction of this work is that such circular references may only point to nodes labeled syntactically with a marker. Since the number of markers is finite, this language precludes the modeling of e.g. doubly-linked lists or leaf-linked trees, but allows root-linked trees.

The ADDS notation in [6] allows the description of abstract properties of pointer structures through the concepts of *dimensions* and *directions*. The main motivation is to make static analysis more feasible through (non-invasive) program annotations. With the ADDS notation one cannot specify the exact shape of values, and manipulations still rely on explicit pointer operations.

The techniques for evaluating routing fields are similar to algorithms for reevaluating attributed grammars [9], but to our knowledge the algorithms for updating a tree of a grammar whose attributes are nodes in the tree has not been described before.

Acknowledgments

Thanks to the anonymous referees for their helpful comments.

References

[1] H. Aït-Kaci and R. Nasr. Logic and inheritance. In *Proc. 13th ACM Symp. on Princ. of Programming Languages*, pages 219–228, 1986.

[2] H. Aït-Kaci and R. Nasr. Login: A logic programming language with built-in in-

heritance. *Journal of Logic Programming*, 3:185–215, 1986. Journal version of [1].

[3] H. Aït-Kaci and A. Podelski. Towards a meaning of life. In Jan Maluszyński and Martin Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (Passau, Germany)*, pages 255–274. Springer-Verlag, LNCS 528, August 1991.

[4] B. Courcelle. The monadic second-order logic of graphs I. Recognizable sets of finite graphs. *Information and computation*, 85:12–75, 1990.

[5] J. Dörre and W.C Rounds. On subsumption and semiunification in feature algebras. In *Proc. IEEE Symp. on Logics in Computer Science*, pages 300–310, 1990.

[6] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proc. SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 249–260. ACM, 1992.

[7] C.A.R. Hoare. Recursive data structures. *International Journal of Computer and Information Sciences*, 4:2:105–132, 1975.

[8] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[9] T. Reps. Incremental evaluation for attribute grammars with unrestricted movement between tree modifications. *Acta Informatica*, 25, 1986.

[10] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 1–16. Springer-Verlag (*LNCS* 201), 1985.

Appendix: Formal Definitions

This appendix contains the formal definitions of the concepts introduced. They may be used to elucidate and substantiate the contents of the preceding summary.

A1: Data Types

Associated with a data type $\mathcal{D}$ we have some notation. The main type is denoted $\mathbf{Main}\,\mathcal{D}$. By $\mathbf{T}_\mathcal{D}$ we denote the set of types. By $\mathbf{T}_\mathcal{D}(T\!:\!v)a$ we denote the type of the data field a in variant v of type T, i.e., for the type-variant above, $\mathbf{T}_\mathcal{D}(T:v)a_i = T_i$. By $\mathbf{V}_\mathcal{D}$ we denote the set of all variants in $\mathcal{D}$; by $\mathbf{V}_\mathcal{D}T$ we denote the set of variants of type T. By $\mathbf{F}_\mathcal{D}$ we denote the set of all data fields in $\mathcal{D}$; by $\mathbf{F}_\mathcal{D}(T:v)$ we denote the set of data fields of type T and variant v, i.e., for the type-variant declaration above, $\mathbf{F}_\mathcal{D}(T:v) = \{a_1, \ldots, a_n\}$. An *address* α is an element of $\mathbf{F}^*_\mathcal{D}$.

The values of $\mathcal{D}$ is the set $\mathbf{Val}\,\mathcal{D}$ of functions $x : \mathbf{F}^*_\mathcal{D} \to \mathbf{T}_\mathcal{D} \times \mathbf{V}_\mathcal{D}$ such that

- $\mathbf{dom}\,x$ is finite and prefix closed;
- $x(\epsilon) = (\mathbf{Main}\,\mathcal{D}\!:\!v)$, for some v; and
- for all $\alpha \in \mathbf{dom}\,x$, if $x(\alpha) = (T\!:\!v)$ then
  - $v \in \mathbf{V}_\mathcal{D}T$ and
  - $\alpha a \in \mathbf{dom}\,x \;\Leftrightarrow\; a \in \mathbf{F}_\mathcal{D}(T:v) \;\wedge\; \mathbf{T}_\mathcal{D}(T\!:\!v)\,a = T'$
    where $x(\alpha a) = (T'\!:\!v')$ for some v'.

Intuitively, the addresses in $\mathbf{dom}\,x$ serve as pointer values.

A2: Graph Types and Routing Expressions

While $\mathbf{F}_\mathcal{G}$ still denotes all fields, we use $\mathbf{F}^\mathbf{d}_\mathcal{G}$ to denote the *data fields*, and $\mathbf{F}^\mathbf{r}_\mathcal{G}$ to denote the *routing fields*. We use the notation $\mathbf{R}_\mathcal{G}(T\!:\!v)\,a$ to denote the routing expression associated

with the routing field a in variant v of type T.

The graph type has an underlying data type $\mathbf{Data}\,\mathcal{G}$ which is obtained by removing all the routing fields. The routing expressions must all be defined on $\mathbf{Data}\,\mathcal{G}$, as described below.

Given a data type $\mathcal{D}$, define the *alphabet* Δ that consists of *directives* (letters) $\wedge$; \$; $\uparrow$; $\uparrow a$ and $\downarrow a$, where $a \in \mathbf{F}_{\mathcal{D}}$; T and $(T:v)$, where $T \in \mathbf{T}_{\mathcal{D}}$ and $v \in \mathbf{V}_{\mathcal{D}} T$.

Given $x \in \mathbf{Val}\,\mathcal{D}$ we define the *step relation* $\rightsquigarrow_x$ on $\mathbf{dom}\,x \times \Delta \times \mathbf{dom}\,x$ by the following transitions:

$$
\begin{array}{ll}
\epsilon \overset{\wedge}{\rightsquigarrow}_x \epsilon & \\
\alpha \overset{\$}{\rightsquigarrow}_x \alpha & \text{if } \alpha \text{ is a leaf in } x \\
\alpha \cdot a \overset{\uparrow}{\rightsquigarrow}_x \alpha & \\
\alpha \cdot a \overset{\uparrow a}{\rightsquigarrow}_x \alpha & \\
\alpha \overset{\downarrow a}{\rightsquigarrow}_x \alpha \cdot a & \\
\alpha \overset{T}{\rightsquigarrow}_x \alpha & \text{if } x(\alpha) = (T:v) \text{ for some } v \\
\alpha \overset{(T:v)}{\rightsquigarrow}_x \alpha & \text{if } x(\alpha) = (T:v)
\end{array}
$$

When $\alpha \overset{d}{\rightsquigarrow}_x \beta$, we say that β is *reached* from α by directive d. Note that β such that $\alpha \overset{d}{\rightsquigarrow}_x \beta$ is uniquely defined, if it exists, by the values of α and d.

A *route* $\rho = d_1 \cdots d_n$ is a word over Δ. A *walk* in x *from* $\alpha \in \mathbf{dom}\,x$ *to* $\beta \in \mathbf{dom}\,x$ *along* ρ is the unique sequence, if it exists, $\alpha_0, \cdots \alpha_n = \beta$, such that $\alpha_{i-1} \overset{d_i}{\rightsquigarrow}_x \alpha_i$ for all i, $1 \le i \le n$. The walk is denoted $\alpha \overset{\rho}{\rightsquigarrow}_x \beta$.

A *routing expression* R on $\mathcal{D}$ is a regular expression over Δ. We construct regular expressions using operators $+$ (union), $\cdot$ (concatenation), and $*$ (iteration). The regular language defined by R is denoted $L(R)$. Given x, R and an *origin* $\alpha \in \mathbf{dom}\,x$, a *destination* is a $\beta \in \mathbf{dom}\,x$ such that $\alpha \overset{\rho}{\rightsquigarrow}_x \beta$ for some route $\rho \in L(R)$. The set of all destinations is denoted $\mathbf{Dest}_x(R, \alpha)$. If this set is a singleton we say that R at α in x has the *unique destination property*.

Intuitively, the routing expressions specify where the pointers in he routing fields should lead to. A graph type is only *well-formed* when all such expressions always have the unique destination property and always lead to subtrees of the specified types.

The values of a well-formed graph type $\mathcal{G}$ form the set $\mathbf{Val}\,\mathcal{G}$ of finite graphs. There is a graph for every value in the underlying data type. Given $x \in \mathbf{Val}\,\mathbf{Data}\,\mathcal{G}$ we construct a graph whose nodes are $\mathbf{dom}\,x$, the set of addresses in x. The edges, which are labeled by field names, come in two flavors: *data edges* and *routing edges*. The data edges provide the canonical spanning tree—the backbone—and are defined as

$$\{\alpha \xrightarrow{a} \alpha a \mid \alpha a \in \mathbf{dom}\,x\}.$$

The routing edges are defined as

$$
\begin{array}{l}
\{\alpha \xrightarrow{a} \beta \mid \\
\quad a \in \mathbf{F}^{\mathbf{r}}_{\mathcal{G}} x(\alpha),\ \mathbf{R}_{\mathcal{G}} x(\alpha) a = R, \\
\quad \mathbf{Dest}_x(R, \alpha) = \{\beta\}\ \}
\end{array}
$$

In this graph, addresses in $\mathbf{F}^*_{\mathcal{G}}$ (both data and routing fields) are defined.

A3: Evaluating Routing Fields

Here we give the details of the algorithm mentioned in Section 5. We are given a backbone x and a collection of nondeterministic finite-state automata representing all routing expressions in the graph grammar. For an automaton A with transition relation $\rightarrow_A$ and a word $w = d_0 \cdots d_n \in \Delta^*$, we write $q \overset{w}{\rightarrow}_A q'$ to denote that there exists $q_0, \ldots, q_{n+1}$ such that $q_0 = q$, $q_{n+1} = q'$, and $q_0 \overset{d_0}{\rightarrow} q_1 \cdots \overset{d_n}{\rightarrow} q_{n+1}$.

Our goal is to build a table Tbl such that for each node α in x and for each automaton A and each state q of A, the value of $Tbl(\alpha, q)$ is a node β, if it exists, such that for some $w \in \Delta^*$, $\alpha \overset{w}{\rightsquigarrow}_x \beta$ and $q \overset{w}{\rightarrow}_A q^F$, where q^F is a final state of A; if no such node exists then $Tbl(\alpha, q) = \text{nil}$.

The algorithm below employs a queue Q to calculate Tbl:

1. $Tbl(\alpha, q) := \text{nil}$, for all nodes α in x and all automata states q

2. make Q empty

3. for all (α, q), where q is a final state:

   (a) $Tbl(\alpha, q) := \alpha$

   (b) insert (α, q) in Q

4. while Q is non-empty:

   (a) delete an element (α, q) from Q

   (b) for all (β, q') such that $Tbl(\beta, q') =$ nil and for some d, $q' \xrightarrow{d}_A q$ and $\beta \overset{d}{\rightsquigarrow}_x \alpha$:

      i. $Tbl(\beta, q') := Tbl(\alpha, q)$

      ii. insert (β, q') in Q

Note that each entry (α, q) is considered at most once and that Step 4.(b) involves only the node α and its immediate neighbors—thus a number of nodes that depends on the grammar only. We conclude that the algorithm runs in linear time as a function of the size of x.

With the well-formedness criterion it is not hard to see that the destination of a routing field at α is the node β if and only if there exists an initial state q of the corresponding automaton such that $Tbl(\alpha, q) = \beta$.

A4: Monadic Logic

The *monadic second-order logic of graph types*, denoted **M2LGT**, is used to express certain properties of graph types. We first introduce a simpler logic, *monadic second-order logic of data types*, denoted **M2LDT**. Fix a data type $\mathcal{D}$. We define the **M2LDT** on $\mathcal{D}$ as follows. There are two kinds of second-order variables, *value variables* and *address set variables*. A value variable x denotes a value of $\mathcal{D}$. An address set variable M denotes a set of addresses of $\mathcal{D}$. Such variables can be combined with $\cup$, $\cap$ and $\emptyset$ to form *address set expressions*. The set of addresses of x is denoted $\mathbf{dom}\, x$, which is also a set expression.

A first-order variable α, also called an *address variable*, denotes an address of $\mathcal{D}$. That α is an address in M is expressed as the formula $\alpha \in M$. A value variable x of type $\mathcal{D}$ is introduced by an existential quantification $\exists_{\mathcal{D}} x$ or a universal quantification $\forall_{\mathcal{D}} x$. Variables that denote addresses or sets of addresses are introduced by usual existential ($\exists$) or universal ($\forall$) quantification. The formulas of the logic are obtained by combining quantification, $\wedge$ (and), $\vee$ (or), $\neg$ (negation) with the following basic formulas:

$\mathbf{is}\wedge(\alpha)$	$\alpha = \epsilon$
$\mathbf{is}_x\$(\alpha)$	$x(\alpha)$ is a leaf variant
$\mathbf{is}_x(T\!:\!v)(\alpha)$	$x(\alpha) = (T : v)$
$\mathbf{is}_x T(\alpha)$	$x(\alpha) = (T : v)$ for some v
$\mathbf{is}_x\mathbf{walk}(\alpha, \beta, R)$	$\exists \rho \in L(R) : \alpha \overset{\rho}{\rightsquigarrow}_x \beta$
$\alpha = \beta$	
$\mathcal{E}_1 = \mathcal{E}_2$	
$\mathcal{E}_1 \subseteq \mathcal{E}_2$	
$\alpha \in \mathcal{E}$	
$\alpha = \beta{\cdot}a$	
$\alpha = \beta \mathbin{\dot{x}} a$	$a \in \mathbf{F}_{\mathcal{D}} x(\beta)$ and $\alpha = \beta{\cdot}a$

where $\mathcal{E}$ and the $\mathcal{E}_i$'s are address set expressions. The formulas have the obvious meanings, e.g. $\mathbf{is}_x(T : v)(\alpha)$ is true iff the type-variant at address α in x is $(T\!:\!v)$. The formula $\alpha = \beta \mathbin{\dot{x}} a$ is true if $\alpha = \beta{\cdot}a$ and a is a field of the type variant at α in x.

Expressing well-formedness

The following formula in **M2LDT** expresses that a graph type $\mathcal{G}$ is well-formed:

$$\begin{aligned}\forall_{\mathcal{D}} x : \mathrm{AND}_{T \in \mathbf{T}_{\mathcal{D}},\, v \in \mathbf{V}_{\mathcal{D}} T}\ \mathrm{AND}_{a \in \mathbf{F}^R(T:v)}\\ \forall \alpha \in \mathbf{dom}\, x : \exists! \beta : \mathbf{is}_x(T\!:\!v)(\alpha) \ \Rightarrow\\ \mathbf{is}_x\mathbf{walk}(\alpha, \beta, \mathbf{R}_{\mathcal{D}} x(\alpha) a),\end{aligned}$$

where $\mathcal{D} = \mathbf{Data}\,\mathcal{G}$ is the underlying data type; $\exists!$ is an abbreviation for "there exists a unique"; and AND is an abbreviation expressing the conjunction obtained by expanding over the corresponding indices.

Decidability of M2LDT

Theorem 1 **M2LDT** *is decidable.*

Proof M2LDT is decidable by an easy reduction to **M2LkSFT**, the monadic second-order logic of k successors on finite trees. The latter logic has *set variables*, such as X, denoting subsets of $\{1,\cdots,k\}^*$ and *first-order variables*, such as α, denoting elements of $\{1,\cdots,k\}^*$. In addition there is a successor function $\cdot k$ for each $j \in \{1,\ldots,k\}$ and connectives and quantifiers as above. We will indicate how formulas of **M2LDT** involving a data type $\mathcal{D}$ can be translated into **M2LkSFT**. We let k be $|\mathbf{F}_\mathcal{D}|$, the number of different fields in $\mathcal{D}$, and we rename field names as $1,\ldots,k$. An x in $\mathcal{D}$ introduced by a quantified formula $\exists_\mathcal{D} x : f$ is translated into $\exists X_d, X_T^1,\ldots,X_T^{n_T}, \exists X_V^1,\ldots,X_V^{n_v} : g \wedge \hat{f}$, where X_d expresses $\mathbf{dom}\, x$; $X_T^1,\ldots X_T^{n_T}$ expresses the type at position α by the bit pattern $\langle \alpha \in X_T^1,\ldots,\alpha \in X_T^{n_T}\rangle$ (here $n_T = \log|\mathbf{T}_\mathcal{D}|$); $X_v^1,\ldots X_v^{n_v}$ expresses the variant at position α by the bit pattern $\langle \alpha \in X_T^1,\ldots,\alpha \in X_T^{n_v}\rangle$ (here $n_v = \log|\mathbf{V}_\mathcal{D}|$); $\hat{f}$ is the translation of f; and g is a formula expressing that x is a value of $\mathcal{D}$ according to the conditions on derivation trees given in Section 1. Address set variables are just translated into set variables and address variables into first-order variables. Most of the basic formulas are now easy to express. For example, $\alpha = \beta\ \dot{x}\ a$ is translated into $\alpha = \beta{\cdot}a\ \wedge\ \alpha \in x$; this formula is equivalent $\alpha = \beta{\cdot}a\ \wedge\ a \in \mathbf{F}_\mathcal{D} x(\beta)$ since $x \in \mathbf{Val}\,\mathcal{D}$. The basic formula $\mathbf{is}_x\mathbf{walk}(\alpha,\beta,R)$ is more difficult. Here we encode the working of A_R, the automaton equivalent to R, on x by a formula that guesses the subsets of states at each α that are accessible from a *partial run* (which is like a run except that the last state need not be final) starting at α. This collection of subsets can be coded using $|A_R|$ set variables. We must then write a **M2LkSFT** formula expressing that all states in a subset have a predecessor for some directive under the transition relation (unless the state is initial and in the subset at α). This alone is not sufficient. We must also write down a condition that ensures that the collection of subsets is minimal with respect to the previous condition; technically, we are calculating a least fixed-point in order to ensure that all states are reachable from initial states at α. The details of this translation are omitted. □

Logic of graph types

The monadic second-order logic of graph types, **M2LGT**, has the same syntax as **M2LDT**.

Theorem 2 **M2LGT** *is decidable.*

Proof The translation into **M2LkSFT** only differs for the formula $\alpha = \beta\ \dot{x}\ a$. If $a \in \mathbf{F}_\mathcal{G}^R x(\beta)$, then the translation must expresses that $\mathbf{is}_x\mathbf{walk}(\beta,\alpha,R)$, where $R = \mathbf{R}\mathcal{G}x(\beta)a$. We omit the details. □