Type Inference for Sublinear Space Functional Programming

Ugo Dal Lago[1]* and Ulrich Schöpp[2]**

[1] University of Bologna, Italy
[2] LMU Munich, Germany

Abstract. We consider programming language aspects of algorithms that operate on data too large to fit into memory. In previous work we have introduced IntML, a functional programming language with primitives that support the implementation of such algorithms. We have shown that IntML can express all LOGSPACE functions but have left open the question how easy it is in practice to program typical LOGSPACE algorithms in IntML. In this paper we develop algorithms for IntML type inference. We show that with type inference one can handle programs that could not be reasonably manipulated by hand. We do so by implementing in IntML a typical LOGSPACE algorithm, a test for acyclicity of undirected graphs. Thus we show that with type inference IntML can express typical algorithmic patterns of LOGSPACE easily and in a natural way.

The study of algorithms operating on data too large to fit into memory has a long tradition in complexity theory; it comprises in particular the complexity classes LOGSPACE and NLOGSPACE of algorithms with logarithmic space usage. By comparison, questions about the programming language aspects of these classes have received little study. What is a good way of programming LOGSPACE algorithms in, say, a functional language? How should one represent values that do not fit in memory and that can only be accessed piece by piece from some external store, such as the input and output of a LOGSPACE function?

To approach such questions, we have recently introduced the functional programming language IntML with support for working with externally stored data [6]. As a first test of the expressive power of IntML, we have shown that each LOGSPACE-function can be expressed in IntML. However, since this result is obtained by encoding Turing Machines, it is not very informative about whether or not it is possible to program such functions easily and in a natural way. In this paper we ask how easy it is to express well-known algorithms with logarithmic space usage in IntML. We take a typical LOGSPACE-algorithm, Cook & McKenzie's test for acyclicity in undirected graphs [5], and show that it can be programmed in IntML in a natural way.

The main contribution of this paper is the development of type inference algorithms for IntML, which make programming such typical LOGSPACE-algorithms in IntML practical. Type inference is particularly important, as the IntML type system allows one to

* The author is partially supported by PRIN project "CONCERTO" and FIRB grant RBIN04M8S8, "Intern. Inst. for Applicable Math."

** Part of this work was carried out while the author was supported by the Institute of Advanced Studies at the University of Bologna.

read off bounds on the space-usage of programs directly from a typing derivation, which means that writing out all type information fully will soon become unmanageable. In this paper we show that much of this information can be reconstructed automatically using a type inference algorithm. While type inference has been shown to be possible for languages capturing PTIME [2, 4], in the context of languages for sublinear space the possibility of type inference is novel.

1 Programming with Bidirectional Data Flow

The definition of IntML can be summarised as follows: start with a standard functional programming language and extend it with primitives for writing programs with bidirectional data flow. Bidirectional data flow is an important property of computation with external data that may not fit into memory. An input that is too large to be stored cannot be read all at once; it can only be queried piece-by-piece during the course of the computation. Thus, information flows not just from the input to a program but in the form of queries also in the opposite direction. We believe that providing a good account for this kind bidirectional data flow should be a central goal in the construction of programming languages for computation with external data.

The construction of IntML starts with a very simple language having just polynomial types α, 1, $A + A$ and $A \times A$. It has the usual terms for these types and also a loop construct for possibly nonterminating iteration. More details are not needed for now (but they appear below). Richer languages would be possible, but this one is suitable for capturing LOGSPACE. We consider this language as a model of computation with unidirectional data flow. A term $c{:}A \vdash f : B$ represents a function from values of type A to values of type B, which can be computed by substituting an input for c and subsequent reduction.

Suppose now that there is some externally stored data that can be queried piece-by-piece. Its interface is given by a pair of types (X^-, X^+). The type X^- consists of the queries that may be sent to the datum and the type X^+ encodes possible answers. For example, to represent binary words that can be queried character-by-character, one may take X^- to be a type representing the character positions in the word, e.g. $X^- = (1 + 1) \times \cdots \times (1 + 1)$, and one may take X^+ to be the type $(1 + 1)$ that represents the two possible characters. We can note here already that the size of the values of these types X^- and X^+ is logarithmic in the length of the words being represented.

A program with bidirectional data flow can now be considered as a message passing node with a number of input and output wires, e.g. Fig. 1. Each wire has a direction and is labelled with a pair of types $X = (X^-, X^+)$. The intention is that values of type X^+ can flow as messages along wires in the direction of the wire, and values of type X^- can flow in the opposite direction, see Fig. 2. Message passing nodes are stateless and may react to messages arriving (one at a time) at one of the wires connected to them. When a message arrives, the node uses it as input in order to compute an output message that it may then pass along one of the wires connected to it. We emphasise that wires are bidirectional and that messages can be passed in *both* directions on all wires. The direction of the wire merely determines which type of message can be passed

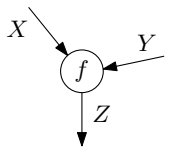


Fig. 1. A node

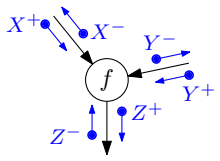


Fig. 2. Types of messages

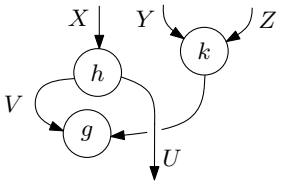


Fig. 3. A circuit

in each direction. It would make no difference if we reversed any edge labelled with $Z = (Z^-, Z^+)$ and changed the label to $Z^* = (Z^+, Z^-)$.

Several such message passing nodes can be combined easily, simply by connecting matching wires to form message passing circuits. For example, in the example circuit in Fig. 3, a value of type U^- may be passed to h against the direction of the output wire. With this input, the node h may then perhaps decide to pass a value of type V^+ along the wire to g or to return a value of type X^- to the environment. It may also decide not to do anything, in which case the whole computation will be blocked. Also, message passing inside the network may go on indefinitely in an infinite loop.

We have found that this way of modelling bidirectional computation by message passing nodes is useful for constructing programs that access externally stored data by means of message passing. This model accounts for sublinear space computation, as the accessed data may be much larger than the messages used to access it.

For the definition of IntML it is important to notice that the message passing circuits can be implemented in the simple functional language that we started with. The node f shown in Fig. 1 can be implemented by a term that takes as input a value of type $Z^- + (X^+ + Y^+)$ and that gives as output a value of type $Z^+ + (X^- + Y^-)$. If a message z of type Z^- arrives at the node, we give $\mathsf{inl}(z)$ as an input to this term and evaluate the result. If the result is $\mathsf{inr}(\mathsf{inl}(x))$, say, then we pass on x along the edge labelled with X. It is not hard to see, even with minimal assumptions about the language, that if all the nodes in a circuit can be implemented thus, then so can the whole circuit. For instance, the circuit in Fig. 3 can be implemented by a term $c\colon U^- + X^+ + Y^+ + Z^+ \vdash u\colon U^+ + X^- + Y^- + Z^-$ that takes as input a message that may arrive at one of the ports of the circuit and that returns the message that will come out of the circuit. Thus, we may think of the circuits as a tool for writing programs with bidirectional data flow in the simple programming language. However, writing such programs for circuits can be quite awkward. Try, for example, to write out a program (in OCaml or Haskell, say) for the circuit in Fig. 3 from given programs for g, h and k.

IntML provides programming language primitives for a more convenient construction and manipulation of such circuits. These primitives are not just a textual representation of circuits; they take the form of well-established programming language constructs like pairs and functions. The new primitives are a conservative extension of the original language; they can only be used to program circuits that could have been programmed directly as well.

$$\frac{}{\Sigma,\, c{:}A \vdash c\colon A} \qquad \frac{}{\Sigma \vdash \mathit{min}_A\colon A} \qquad \frac{\Sigma \vdash f\colon A}{\Sigma \vdash \mathit{succ}_A(f)\colon A} \qquad \frac{\Sigma \vdash f\colon A \qquad \Sigma \vdash g\colon A}{\Sigma \vdash \mathit{eq}_A(f,g)\colon 1+1}$$

$$\frac{}{\Sigma \vdash *\colon 1} \qquad \frac{\Sigma \vdash f\colon A \qquad \Sigma \vdash g\colon B}{\Sigma \vdash \langle f,g\rangle\colon A \times B} \qquad \frac{\Sigma \vdash f\colon A \times B}{\Sigma \vdash \mathsf{fst}(f)\colon A} \qquad \frac{\Sigma \vdash f\colon A \times B}{\Sigma \vdash \mathsf{snd}(g)\colon B}$$

$$\frac{\Sigma \vdash f\colon A}{\Sigma \vdash \mathsf{inl}(f)\colon A+B} \qquad \frac{\Sigma \vdash f\colon A+B \qquad \Sigma,\, c{:}A \vdash g\colon C \qquad \Sigma,\, d{:}B \vdash h\colon C}{\Sigma \vdash \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow g \mid \mathsf{inr}(d) \Rightarrow h\colon C}$$

$$\frac{\Sigma \vdash f\colon B}{\Sigma \vdash \mathsf{inr}(f)\colon A+B} \qquad \frac{\Sigma,\, c{:}A \vdash f\colon A+B \qquad \Sigma \vdash g\colon A}{\Sigma \vdash \mathsf{loop}(c.\, f)(g)\colon B} \qquad \frac{\Sigma \mid \vdash t\colon [A]}{\Sigma \vdash \mathsf{unbox}(t)\colon A}$$

Fig. 4. Working Class Typing Rules

2 IntML

IntML has two classes of terms and types, one corresponding to the functional programming language that we start with and one for working with circuits over this language. We call the former the *working class* and the latter the *upper class*. This terminology reflects that all computation will be done by working class terms. Upper class terms merely represent message passing circuits; the actual message passing will be done by the working class terms implementing the circuits.

For completeness we include a definition of the working class calculus. Its types are

$$A ::= \alpha \mid 1 \mid A + A \mid A \times A$$

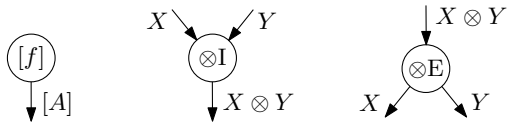
and the typing rules are given in Fig. 4. This language is a fairly standard call-by-value language. For convenience we include constants min_A, succ_A and eq_A for any type, including types with variables. These constants provide a total ordering on any type generically. For example, min_{1+1} is $\mathsf{inl}(*)$ and its successor is $\mathsf{inr}(*)$. The constant loop is a simple way of including iteration in the language. The intended operational semantics is $\mathsf{loop}(c.f)(v) \longrightarrow \mathsf{case}\ f[v/c]\ \mathsf{of}\ \mathsf{inl}(d) \Rightarrow \mathsf{loop}(c.f)(d) \mid \mathsf{inr}(d) \Rightarrow d$.

The upper class part of IntML is a calculus of circuits over working class terms. Upper class terms denote circuits and the types denote the labels of the wires going in and coming out of such circuits.

The upper class types in IntML are formed by the following grammar.

$$X ::= \beta \mid [A] \mid X \otimes X \mid A \cdot X \multimap X$$

The type $[A]$ is intended to denote a wire with $[A]^- = 1$ and $[A]^+ = A$. It represents an interface of a thunk: we may send the unique element of 1 as a signal to start some computation, whose result of type A we expect to arrive from the same wire. A thunk node $[f]$, as shown below, can be implemented by a working class term $c{:}1 \vdash f\colon A$. The type $X \otimes Y$ represents a bundle of a wire of type X and a wire of type Y. Such a bundle can be implemented as a single wire with $(X \otimes Y)^- = X^- + Y^-$ and $(X \otimes Y)^+ = X^+ + Y^+$. It is straightforward to implement message passing nodes, as shown below, for the packing and unpacking of such bundles.



The type $A \cdot X \multimap Y$, in which A is a working class type, represents a linear function space. One should read $A \cdot X$ as an A-fold copy of X, one copy for each value of type A. Thus, the type $A \cdot X \multimap Y$ denotes functions from X to Y, in which the argument can be used A-many times. We use a type A instead of natural numbers or polynomials, since we usually want to use working class values to address the particular copies of X and it would be awkward to have to encode and decode them as numbers in the language. We call A the *index type* of the function.

In terms of circuit wires, the type $A \cdot X \multimap Y$ is implemented by $(A \cdot X \multimap Y)^- = A \times X^+ + Y^-$ and $(A \cdot X \multimap Y)^+ = A \times X^- + Y^+$. This definition captures functions in a way familiar from game semantics, see e.g. [1]. When one asks a function for its result (sends a query in Y^-), it may come back with an answer (a message in Y^+) or with a request for its argument (a message in X^-). The argument circuit may send its answer (of type X^+) along the same channel as the original question. The A-fold replication of the argument is implemented by adding to the message a component $A \times -$ that indicates the copy that we are communicating with.

The terms of the upper class calculus represent circuits. The typing sequents have the form

$$\Sigma \mid x_1 : A_1 \cdot X_1, \ldots, x_n : A_n \cdot X_n \vdash t : Y$$

where Σ is a working class context. In this sequent each upper class variable x_i is assigned an upper class type X_i and appears with a multiplicity given by an *index type* A_i, which is a working class type. The term t in this sequent represents a circuit with one outgoing wire labelled Y and with n incoming wires labelled with $A_1 \cdot X_1$, $\ldots$, $A_n \cdot X_n$, where $A_i \cdot X_i = (A_i \times X_i^-, A_i \times X_i^+)$. This circuit is implemented by a working class program that may refer to the variables from Σ.

The upper class calculus appears in Fig. 5. In these rules we use the notation $A \cdot \Gamma$ to denote the context obtained by replacing each declaration $x : B \cdot X$ in Γ by the declaration $x : (A \times B) \cdot X$.

The upper class terms represent a number of constructions on circuits. For example, the term $\langle s, t \rangle$ in rule ($\otimes$I) corresponds to taking the two circuits for s and t and joining their output wires with the node $\otimes$I shown above.

Note that all rules are additive in the context Σ, so that in particular the variables from this context can be used more than once. The only upper class rule that modifies the context Σ is ([ ]E). Informally the let-term in this rule first requests the output from the circuit for s. Upon receipt of this value (of type A), it then binds c to this value and requests the value of circuit t.

The structural rule (STRUCT) has a side condition $A \precsim B$, which informally states that any value of type A can be converted without loss into a value of type B. The rule is sound whenever A is a retract of B, which means that there are working class terms $c{:}A \vdash f : B$ and $d{:}B \vdash g : A$ such that for any closed value v of type A, the term $g[f[v/c]/d]$ reduces to v.

$$\text{(Var)}\ \frac{}{\Sigma \mid \Gamma,\, x : 1{\cdot}X \vdash x : X} \qquad \text{(Struct)}\ \frac{\Sigma \mid \Gamma,\, x : A{\cdot}X \vdash s : Y}{\Sigma \mid \Gamma,\, x : B{\cdot}X \vdash s : Y}\ A \precsim B$$

$$\text{(Weak)}\ \frac{\Sigma \mid \Gamma \vdash s : Y}{\Sigma \mid \Gamma,\, x : A{\cdot}X \vdash s : Y} \qquad \text{(Exch)}\ \frac{\Sigma \mid \Gamma,\, x : A{\cdot}X,\, y : B{\cdot}Y,\, \Delta \vdash s : Z}{\Sigma \mid \Gamma,\, y : B{\cdot}Y,\, x : A{\cdot}X,\, \Delta \vdash s : Z}$$

$$(\otimes\text{I})\ \frac{\Sigma \mid \Gamma \vdash s : X \qquad \Sigma \mid \Delta \vdash t : Y}{\Sigma \mid \Gamma,\, \Delta \vdash \langle s, t\rangle : X \otimes Y}$$

$$(\otimes\text{E})\ \frac{\Sigma \mid \Gamma \vdash s : X \otimes Y \qquad \Sigma \mid \Delta,\, x : A{\cdot}X,\, y : A{\cdot}Y \vdash t : Z}{\Sigma \mid \Delta,\, A \cdot \Gamma \vdash \mathsf{let}\ s\ \mathsf{be}\ \langle x, y\rangle\ \mathsf{in}\ t : Z}$$

$$(\multimap\text{I})\ \frac{\Sigma \mid \Gamma,\, x : A{\cdot}X \vdash s : Y}{\Sigma \mid \Gamma \vdash \lambda x.\, s : A \cdot X \multimap Y} \qquad (\multimap\text{E})\ \frac{\Sigma \mid \Gamma \vdash s : A \cdot X \multimap Y \qquad \Sigma \mid \Delta \vdash t : X}{\Sigma \mid \Gamma,\, A \cdot \Delta \vdash s\ t : Y}$$

$$\text{(Contr)}\ \frac{\Sigma \mid \Gamma \vdash s : X \qquad \Sigma \mid \Delta,\, x : A{\cdot}X,\, y : B{\cdot}X \vdash t : Y}{\Sigma \mid \Delta,\, (A + B) \cdot \Gamma \vdash \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t : Y}$$

$$\text{(Case)}\ \frac{\Sigma \vdash f : A + B \qquad \Sigma, c{:}A \mid \Gamma \vdash s : X \qquad \Sigma, d{:}B \mid \Gamma \vdash t : X}{\Sigma \mid \Gamma \vdash \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow s \mid \mathsf{inr}(d) \Rightarrow t : X}$$

$$([\,]\text{I})\ \frac{\Sigma \vdash f : A}{\Sigma \mid - \vdash [f] : [A]} \qquad ([\,]\text{E})\ \frac{\Sigma \mid \Gamma \vdash s : [A] \qquad \Sigma, c{:}A \mid \Delta \vdash t : [B]}{\Sigma \mid \Gamma,\, A \cdot \Delta \vdash \mathsf{let}\ s\ \mathsf{be}\ [c]\ \mathsf{in}\ t : [B]}$$

$$\text{(Hack)}\ \frac{\Sigma, c{:}X^- \vdash f : X^+}{\Sigma \mid - \vdash \mathsf{hack}_X(c.f) : Y}\ X \precsim Y$$

Fig. 5. Upper Class Typing Rules

In IntML we choose for $\precsim$ the following syntactic approximation of retraction. Let *structural congruence* $\cong$ be the smallest congruence relation on working class types that satisfies:

$$A \times 1 \cong A \qquad 1 \times A \cong A \qquad \text{(U)}$$
$$A \times (B + C) \cong A \times B + A \times C \qquad (B + C) \times A \cong B \times A + C \times A \qquad \text{(D)}$$

Let $\leq$ be the the least reflexive, transitive relation that satisfies

$$A \leq A + B \qquad B \leq A + B \qquad A \leq A \times B \qquad B \leq A \times B$$

and that is compatible with all type operations, which means that $A \leq B$ implies $C[A/\alpha] \leq C[B/\alpha]$ for all C. With these definitions, we define

$$A \precsim B \iff \exists C.\, A \leq C \cong B.$$

We remark that $A \precsim B$ is also equivalent to $\exists C, D.\, A \cong C \leq D \cong B$ and moreover that $\precsim$ is transitive.

Lemma 1. *If $A \precsim B$ and $B \precsim A$ then $A \cong B$.*

We include the rule (STRUCT) to make index types more flexible. The unit laws (U) allow us to give the composition $\lambda x.\, f\ (g\ x)$ of two upper class functions $f\colon 1 \cdot X \multimap Y$ and $g\colon 1 \cdot Y \multimap Z$ the type $1 \cdot X \multimap Z$. Without it this term can only be given type $(1 \times 1) \cdot X \multimap Z$. Distributivity (D) is useful to type the terms $(s\ (\mathsf{copy}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ t))$ and $(\mathsf{copy}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ (s\ t))$ in the same context. Without distributivity we only have:

$$\Sigma \mid \Gamma,\, x\colon (A \times (C + D)) \cdot X \vdash s\ (\mathsf{copy}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ t)\colon Z$$
$$\Sigma \mid \Gamma,\, x\colon (A \times C + A \times D) \cdot X \vdash \mathsf{copy}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ (s\ t)\colon Z$$

Finally, the inequalities $\leq$ are often useful in conjunction with rule ($\otimes$E). The right premise of ($\otimes$E) demands that the index types of x and y are the same and the inequalities can be used to satisfy that demand.

The upper class rules are completed by the rule (HACK). With this rule a programmer can implement message passing nodes directly by providing a working class term. This rule can be used to define constants of complex types that would otherwise not be definable. We like to think of (HACK) as an analogue of inline assembly in C.

An important use of (HACK) is for the definition of an iteration combinator [6]:

$$\mathtt{loop}\colon \alpha \cdot (\gamma \cdot [\alpha] \multimap [\alpha + \beta]) \multimap [\alpha] \multimap [\beta]$$

Informally, the first argument of `loop` is the step function and the second argument is the initial value. The return value of the step function is either the final result (of type β) or the value for the next application of the step function: $\mathtt{loop}\ s\ t = [w]$ if $s\ t = [\mathsf{inr}(w)]$ and $\mathtt{loop}\ s\ t = \mathtt{loop}\ s\ [w]$ if $s\ t = [\mathsf{inl}(w)]$. To make loops easier to read, we use $\mathsf{return}(w)$ and $\mathsf{continue}(w)$ as abbreviations for $\mathsf{inr}(w)$ and $\mathsf{inl}(w)$ respectively.

We formulate the rule (HACK) so that the type of $\mathsf{hack}_X(c.f)$ is closed under structural manipulation analogous to rule (STRUCT). Instead of requiring $\mathsf{hack}_X(c.f)$ to have type X, we allow any type Y with $X \precsim Y$, where $\precsim$ is extended to upper class types as follows:

$$\begin{aligned}
[A] \precsim [A] &\iff \top \\
(X \otimes Y) \precsim (Z \otimes U) &\iff (X \precsim Z) \wedge (Y \precsim U) \\
(A \cdot X \multimap Y) \precsim (B \cdot Z \multimap U) &\iff (B \precsim A) \wedge (Z \precsim X) \wedge (Y \precsim U)
\end{aligned}$$

Note that $X \precsim Y$ can only hold if X does not contain upper class type variables.

For details about the translation of upper class terms to circuits we refer to [6]. For the practical use of the upper class language, it is not important to know the technical details of the translation. The upper class terms may be understood in terms of a simple operational semantics with reductions such as $(\lambda x.\, s)\ t \longrightarrow s[t/x]$ and $(\mathsf{let}\ [v]\ \mathsf{be}\ [c]\ t) \longrightarrow t[v/c]$ and the translation to circuits is a sound (and space efficient!) implementation of this operational semantics [6].

Nevertheless, we should outline how the translation to circuits can be used to obtain space bounds for IntML programs. An upper class IntML-term is compiled to a circuit in which each wire is annotated with a pair of working class types. The working class types that thus appear in the compiled circuits can be read-off from a typing derivation

of the term. Now, in the evaluation of a circuit, at any time one needs to store only one single value, namely the message that is just being passed along some edge, together with its position. Hence, the space needed to evaluate a circuit depends only on the size of the circuit and the maximum size of a message that can be passed along one of its edges. The size of the circuit is constant and a bound on the size of messages can be found simply by looking at the types that are written on the various wires. Due to the simplicity of the working class calculus, it is quite easy to give bounds on the size of values of given working class types. In this way, we obtain useful bounds on the space usage of IntML programs simply by looking at the types that appear in the circuits of upper class programs.

3 Application: Graph Algorithms

To illustrate that algorithms with external data can be expressed quite naturally in IntML and to show how interesting space bounds can be read off from a typing derivation, we present an implementation of a classic LOGSPACE graph algorithm in IntML. We implement the test of acyclicity in undirected graphs due to Cook & McKenzie [5].

We use a higher-order representation of graphs, where a graph is given by two predicates on a carrier set. The first predicate is unary and tells which elements of the carrier represent graph nodes; and the second predicate is binary and encodes the edge predicate. In IntML such a pair of predicates can be represented by the upper class type below. Therein, α is the carrier and β and γ are parameters (which we will usually elide).

$$\mathit{Graph}_{\beta,\gamma}(\alpha) = (\beta \cdot [\alpha] \multimap [2]) \otimes (\gamma \cdot [\alpha \times \alpha] \multimap [2])$$

Since in IntML each working class type comes with a total ordering, we do not need to choose up front a type for α and can just use the total ordering on this type variable α.

Leaving α to be a type variable is important also in order to handle input graphs of arbitrary size. Suppose we have a program of polymorphic type $A \cdot \mathit{Graph}(\alpha) \multimap X$ and we have a (large, externally stored) graph of size n that we would like to give as input to that program. To do this we choose for α a type N large enough to encode all the nodes of the graph. The interface between the program and the graph is then given in terms of the messages that are being passed along the edge with label $(A[N/\alpha]) \cdot \mathit{Graph}(N)$.

The choice of graph representation is not only quite natural, it also allows us to obtain logarithmic space bounds easily. If for α we choose the type $2 \times \cdots \times 2$ (k times), where $2 = 1 + 1$, then $\mathit{Graph}(\alpha)$ can represent graphs with up to 2^k nodes. On the other hand, the values of type $\mathit{Graph}(\alpha)^-$ and $\mathit{Graph}(\alpha)^+$ have size $O(k)$, as is easily seen directly. This simple observation will allow us to obtain logarithmic space bounds (see [6] for further details).

The algorithm of Cook & McKenzie for checking acyclicity of undirected graphs can be explained using the notion of a right-hand walk. A right-hand walk is like a walk in a labyrinth where one always keeps his right hand on the wall. One imagines the graph edges to be corridors and the nodes to be junctions. The edges connected to each node are ordered as if the node were a junction of the arriving corridors. Cook & McKenzie's observation then is that an undirected graph is acyclic if and only if any right-hand walk will return to its starting point by traversing the edge over which it has

left this point in the opposite direction. This observation leads to a LOGSPACE algorithm, since this property can be checked by following right-hand walks, and for this one needs to keep in memory just a few graph nodes.

To present an implementation of this algorithm in IntML, we introduce a few notational abbreviations. First, when programming in IntML it is usually not necessary to think about the index types. We will therefore hide them and write just $X \to Y$ instead of $A \cdot X \multimap Y$, with the understanding that A is still there, it is just not shown.

For accessing the node and edge predicates of a given graph we use functions *node* and *edge* defined by

$$\begin{aligned} \mathit{node} &= \lambda \mathit{graph}.\, \mathsf{let}\ \mathit{graph}\ \mathsf{be}\ \langle n, e\rangle\ \mathsf{in}\ n, \\ \mathit{egde} &= \lambda \mathit{graph}.\, \mathsf{let}\ \mathit{graph}\ \mathsf{be}\ \langle n, e\rangle\ \mathsf{in}\ e. \end{aligned}$$

We also write *src* and *dst* as abbreviations for fst and snd when they are used on pairs in $\alpha \times \alpha$ that represent edges.

We use 2 as a type of boolean values and define $\mathit{true} = \mathsf{inl}(*)$ and $\mathit{false} = \mathsf{inr}(*)$. Upper class case distinction *if*: $[2] \to X \to X \to X$ can for arbitrary X be defined by $\lambda b.\, \lambda x.\, \lambda y.\, \mathsf{let}\ b\ \mathsf{be}\ [c]\ \mathsf{in}\ \mathsf{case}\ c\ \mathsf{of}\ \mathsf{inl}(t) \Rightarrow x \mid \mathsf{inr}(f) \Rightarrow y$. We also use a similarly defined function *and*: $[2] \to [2] \to [2]$.

Finally, we define a working class term $\mathit{cysucc}_\alpha \colon \alpha \to \alpha$, which is like succ_α but which wraps around to the minimum element when it reaches the maximum.

With these definitions, we can implement an algorithm for checking acyclicity. In order to implement right-hand walks, we first need a function

$$\mathit{nextEdge}_\alpha \colon \mathit{Graph}(\alpha) \to [\alpha \times \alpha] \to [\alpha \times \alpha]$$

that takes an edge $\langle s, d\rangle$ and returns the next edge $\langle s, e\rangle$ emanating from the same source. Starting from $\langle s, \mathit{cysucc}_\alpha(d)\rangle$, this function applies cysucc_α to the second part of the pair until it has found an edge. A definition of *nextEdge* is given in the appendix.

We can then write a function $\mathit{checkpath}_\alpha$, which follows a right-hand-rule walk starting from some given edge and checks if this walk returns to its origin by walking the given edge in the opposite direction.

```
checkpath_α : Graph(α) → [α × α] → [2] =
  λgraph. λinputedge. copy graph as graph1, graph2 in
                      let inputedge be [e] in
                      if (edge graph1 [e])
                        (loop (λw. let w be [p] in
                                   if [eq_α(dst p, src e)]
                                     [return(eq_α(src p, dst e))]
                                     (let nextEdge_α graph2 [⟨dst p, src p⟩]
                                        be [d] in [continue(d)])) [e])
                        [true]
```

Now, writing a function $\mathit{checkcycle}_\alpha$ for testing acyclicity of undirected graphs it a simple matter of applying $\mathit{checkpath}_\alpha$ to all edges in the graph and combining the results

using *and*. We can do this by using a combinator fold_β, which is such that $\mathit{fold}_\beta\ f\ y$ computes $f\ x_n\ (\ldots(f\ x_1\ (f\ x_0\ y)))$, where $x_0 = \mathit{min}_\beta$ and $x_{i+1} = \mathit{succ}_\beta(x_i)$ and x_n is the maximum element of β, i.e. the element with $x_n = \mathit{succ}_\beta(x_n)$. The combinator fold_β is not hard to define, see [6].

$$
\begin{array}{l}
\mathit{checkcycle}_\alpha : \mathit{Graph}(\alpha) \to [2] = \\
\quad \lambda \mathit{graph}.\ \mathsf{copy}\ \mathit{graph}\ \mathsf{as}\ \mathit{graph1}, \mathit{graph2}\ \mathsf{in} \\
\qquad \mathit{fold}_{\alpha\times\alpha}\ (\lambda \mathit{vertexpair}.\ \lambda \mathit{acyclic}. \\
\qquad\qquad\qquad \mathsf{let}\ \mathit{vertexpair}\ \mathsf{be}\ [e]\ \mathsf{in} \\
\qquad\qquad\qquad \mathit{and}\ \mathit{acyclic} \\
\qquad\qquad\qquad\quad (\mathit{if}\ (\mathit{edge}\ \mathit{graph1}\ [e]) \\
\qquad\qquad\qquad\qquad (\mathit{checkpath}_\alpha\ \mathit{graph2}\ [e]) \\
\qquad\qquad\qquad\qquad [\mathit{true}])) \\
\qquad\qquad [\mathit{true}]
\end{array}
$$

Above we have hidden the index types in the type of $\mathit{checkcycle}_\alpha$. Fully spelt out, this function may be given the type

$$A \cdot ((1 \cdot [\alpha] \multimap [2]) \otimes (1 \cdot [\alpha \times \alpha] \multimap [2])) \multimap [2],$$

where A is the working class type $(\alpha \times \alpha \times 2 \times (\alpha \times \alpha \times 2) \times (\alpha \times \alpha \times 2) + \alpha \times \alpha \times 2 \times (\alpha \times \alpha \times 2) \times (\alpha \times \alpha \times (2 \times (2 \times ((\alpha \times \alpha + \alpha \times \alpha \times (2 \times (\alpha \times \alpha \times (\alpha \times \alpha \times (2 \times (\alpha \times \alpha \times (\alpha \times \alpha \times (\alpha \times \alpha)))))))))))))$. This index type may look complicated, but we did not have to consider it when writing the function above. It was computed by a type inference algorithm only *after* the program was already written.

The type A tells us something about the space usage of the program. Whenever the program sends a request to the graph, such as whether some element of type α is a node or whether a pair in $\alpha \times \alpha$ is an edge, it encodes its internal state as a value of type A, so that it can resume work when an answer arrives. Examining A, we see that storing one of its elements needs about as much space to store 20 elements of type α, which here represent graph nodes.

By considering all the edges in the circuit for $\mathit{checkcycle}_\alpha$ in this way, we can read off from the circuit how many graph nodes this function needs to store and thus obtain an upper bound on its space usage. In this example, our prototype implementation obtains a space bound of $20x + 1460$, where x is an upper bound on the size of the values in α. In other words, the program needs to store 20 nodes and it needs some constant space of size 1460. Here, we use the standard size of values, e.g. $|\langle f, g\rangle| = 1 + |f| + |g|$ and $|\mathsf{inl}(f)| = 1 + |f|$. For a given input graph with n nodes, we can take α to be $2 \times \cdots \times 2$ ($\lceil \log n \rceil$ times), so that simply by looking at the types of the circuits we obtain a logarithmic space bound for the overall program.

4 Type Inference

The graph algorithm example shows that it is not hard to understand which type an upper class term has, so long as we hide the index types. We have found that the types

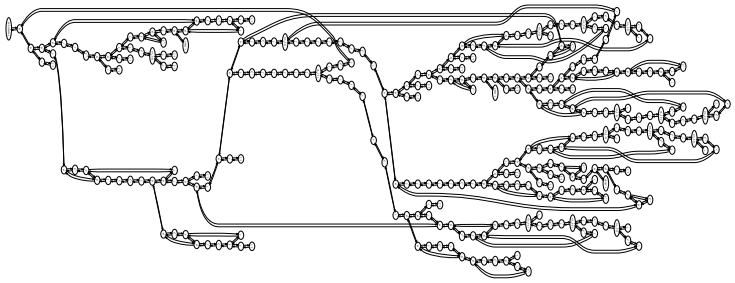


Fig. 6. Overview of the circuit for *checkcycle*$_\alpha$

are not generally a hindrance when writing upper class programs. However, the example also shows that the index types quickly become too complicated to be handled by hand. The practicability of IntML depends on useful type inference algorithms. In this section we analyse the problem of type inference and develop the simple algorithm that we have used to infer the types of the above example.

Type inference is essential to making IntML programming practical: IntML programs have minimal type annotations, but once they have been typed, space bounds for their evaluation can be read off from the types, as explained above for graph algorithms. Thus, type inference amounts to inference of space bounds and we cannot expect a programmer to calculate such bounds by hand.

4.1 Constraint-Based Type Inference

As is standard, see e.g. [9], we separate the type inference algorithms into two parts: finding a set of constraints that needs to be solved in order for a term to be typed and solving the constraint set.

Definition 1. *A* constraint *is either an equality $X = Y$ between upper class types or an equality $A = B$, a structural inequality $A \precsim B$ or a congruence $A \cong B$ between working class types.*

Although $A \cong B$ can be expressed by $\{A \precsim B,\ B \precsim A\}$, we include $\cong$-constraints for technical convenience.

For a type substitution σ, i.e. a finite mapping from upper class type variables to upper class types and working class type variables to working class types, and a set of constraints C, we write $\sigma \models C$ if applying σ to all the types in C makes all the constraints therein true (in the evident sense). We say that σ *is a solution of* C.

We define two partial type inference functions $\mathcal{T}(\Sigma, f)$ and $\mathcal{T}(\Sigma \mid \Gamma, t)$. The first returns a pair (A, C) of a working-class type and a set of constraints, and the second returns a pair (X, C) of an upper class type and a set of constraints. These functions compute principal types in the following sense:

$$\frac{\Gamma \text{ contains } x : A{\cdot}X}{\mathcal{T}(\Sigma \mid \Gamma, x) = (X, \{1 \precsim A\})} \qquad \frac{\mathcal{T}(\Sigma \mid (\Gamma,\, x : \alpha{\cdot}\beta), s) = (X, C) \qquad \alpha, \beta \text{ fresh}}{\mathcal{T}(\Sigma \mid \Gamma, \lambda x.\, s) = (\alpha \cdot \beta \multimap X, C)}$$

$$\frac{\begin{array}{ll}\mathcal{T}(\Sigma \mid \Gamma|_s, s) = (Y_1, C_1) & \\ \Gamma|_t = (x_1 : A_1{\cdot}X_1, \ldots, x_n : A_n{\cdot}X_n) & \vec{\alpha}, \beta, \gamma, \delta \text{ fresh} \\ \mathcal{T}(\Sigma \mid (x_1 : \alpha_1{\cdot}X_1, \ldots, x_n : \alpha_n{\cdot}X_n), t) = (Y_2, C_2) & FV(s) \cap FV(t) = \emptyset\end{array}}{\mathcal{T}(\Sigma \mid \Gamma, s\ t) = (\beta, C_1 \cup C_2 \cup \{Y_1 = \delta \cdot \gamma \multimap \beta, Y_2 = \gamma\} \cup \{\delta \times \alpha_i \precsim A_i \mid 1 \le i \le n\}))}$$

$$\frac{\begin{array}{ll}\Gamma|_s = (x_1 : A_1{\cdot}X_1, \ldots, x_n : A_n{\cdot}X_n) & \\ \mathcal{T}(\Sigma \mid (x_1 : \alpha_1{\cdot}X_1, \ldots, x_n : \alpha_n{\cdot}X_n), s) = (Y_1, C_1) & \vec{\alpha}, \beta, \gamma, \delta \text{ fresh} \\ \mathcal{T}(\Sigma \mid (\Gamma|_t,\, x : \delta{\cdot}\gamma,\, x : \beta{\cdot}\gamma), t) = (Y_2, C_2) & FV(s) \cap FV(t) = \emptyset\end{array}}{\mathcal{T}(\Sigma \mid \Gamma, \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t) = (Y_2, C_1 \cup C_2 \cup \{Y_1 = \gamma\} \cup \{(\delta + \beta) \times \alpha_i \precsim A_i \mid 1 \le i \le n\})}$$

$$\frac{\mathcal{T}(\Sigma, f) = (A, C)}{\mathcal{T}(\Sigma \mid \Gamma, [f]) = ([A], C)}$$

$$\frac{\begin{array}{ll}\mathcal{T}(\Sigma \mid \Gamma|_s, s) = (Y_1, C_1) & \\ \Gamma|_t = (x_1 : A_1{\cdot}X_1, \ldots, x_n : A_n{\cdot}X_n) & \vec{\alpha}, \beta \text{ fresh} \\ \mathcal{T}((\Sigma,\, c{:}\beta) \mid (x_1 : \alpha_1{\cdot}X_1, \ldots, x_n : \alpha_n{\cdot}X_n), t) = (Y_2, C_2) & FV(s) \cap FV(t) = \emptyset\end{array}}{\mathcal{T}(\Sigma \mid \Gamma, \mathsf{let}\ s\ \mathsf{be}\ [c]\ \mathsf{in}\ t) = (Y_2, C_1 \cup C_2 \cup \{Y_1 = [\beta]\} \cup \{\beta \times \alpha_i \precsim A_i \mid 1 \le i \le n\})}$$

Fig. 7. Upper Class Type Inference Rules (selection)

Proposition 1 (Soundness).
1. *If* $\mathcal{T}(\Sigma, f) = (A, C)$ *and* $\sigma \models C$ *then* $\Sigma\sigma \vdash f\sigma : A\sigma$.
2. *If* $\mathcal{T}(\Sigma \mid \Gamma, t) = (X, C)$ *and* $\sigma \models C$ *then* $\Sigma\sigma \mid \Gamma\sigma \vdash t\sigma : X\sigma$.

Proposition 2 (Completeness).
1. *If* $\Sigma\sigma \vdash f\sigma : A\sigma$ *then there exist* B *and* C *with* $\mathcal{T}(\Sigma, f) = (B, C)$ *such that* σ *can be extended to a type substitution* ρ *satisfying* $A\sigma = B\rho$ *and* $\rho \models C$.
2. *If* $\Sigma\sigma \mid \Gamma\sigma \vdash t\sigma : X\sigma$ *then there exist* Y *and* C *with* $\mathcal{T}(\Sigma \mid \Gamma, t) = (Y, C)$ *such that* σ *can be extended to a type substitution* ρ *satisfying* $X\sigma = Y\rho$ *and* $\rho \models C$.

In Fig. 7 we give a selection of typical rules for $\mathcal{T}(\Sigma \mid \Gamma, t)$. These rules formalise a definition of $\mathcal{T}(\Sigma \mid \Gamma, t)$ by structural recursion over t. In the rules we write $\Gamma|_s$ for the subcontext of Γ, in which only the free (term)-variables that appear freely in s are being declared. It is easy to see that (up to the choice of fresh names) these rules define a partial function, which is easy to compute. We omit standard rules for $\mathcal{T}(\Sigma, f)$.

Propositions 1 and 2 can then be proved in a standard way by induction on the derivations of $\mathcal{T}(\Sigma, f) = (A, C)$ and $\mathcal{T}(\Sigma \mid \Gamma, t) = (X, C)$ respectively.

4.2 Solving Constraints

Having reduced type inference to constraint solving, it remains to study how constraint sets can be solved.

Reducing Constraints to E-Unification. First we note that because equality up to congruence $A \cong B$ is a constraint, constraint solving must be at least as hard as E-unification, the problem of unification up to some given equational theory. E-unification is a well-studied problem and there are results for a wide range of equational theories, see [3, 8] for an overview.

Constraint solving can be reduced to E-unification. Equality constraints can be removed up front by standard unification, leaving us with constraints of the form $A \cong B$ and $A \precsim B$. To eliminate the latter, we recall that $A \precsim B$ is equivalent to $\exists C, D.\, A \cong C \leq D \cong B$. This suggests a simple nondeterministic algorithm for reducing constraint solving to E-unification. For any structural constraint $A \precsim B$, we guess type C and D with $C \leq D$ and replace the $A \precsim B$ by $A \cong C$ and $D \cong B$. Now we can use E-unification to solve the remaining constraints.

The efficiency of this approach depends on the maximum size that needs to be taken into account for C and D as well as the efficiency of an E-unification procedure. Constraints like $C \leq D$ can be verified easily, e.g. using dynamic programming.

Unfortunately, to the best of our knowledge, the problem of E-unification for the equational theory of (U)+(D) is still open. Moreover, even if we could find an algorithm for this problem, it would most likely be unpractical for IntML type inference. At the very least it would be NP-hard [10], although it might be much worse.

That constraint solving is hard does not immediately imply the same for IntML type inference. One could hope that in type inference only certain kinds of constraint sets can arise, which are easier to solve. However, this is not the case:

Proposition 3. *Solving the constraint sets that arise in* IntML *type inference is at least as hard as equational unification for (U)+(D).*

Avoiding Distributivity. One choice in the definition of IntML that makes type checking hard is the inclusion of distributivity laws in rule (STRUCT), since in conjunction with the unit laws this appears to make equational unification hard. This leads us to reconsidering the definition of structural congruence. Our motivation for introducing the distributivity laws was to be able to commute copy with other operators without having to change index types in the context, as explained in Sec. 2.

With a reasonable restriction to the IntML type system, we can obtain a similar property without the distributivity laws. We may weaken the contraction rule as follows:

$$(\textsc{Contr2})\ \frac{\Sigma \mid \Gamma \vdash s\colon X \qquad \Sigma \mid \Delta,\, x\colon A{\cdot}X,\, y\colon A{\cdot}X \vdash t\colon Y}{\Sigma \mid \Delta,\, (2 \times A) \cdot \Gamma \vdash \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t\colon Y}$$

This rule is derivable with the distributivity and unit laws, but (CONTR) is not derivable from (CONTR2). With (CONTR2) the two problematic copy-terms can be typed as:

$$\Sigma \mid \Gamma,\, x\colon A \times (2 \times B){\cdot}X \vdash s\ (\mathsf{copy}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ t)\colon Y$$
$$\Sigma \mid \Gamma,\, x\colon 2 \times (A \times B){\cdot}X \vdash \mathsf{copy}\ x\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ (s\ t)\colon Y$$

Therefore, with the weaker version of contraction, we may use the associativity (A) and commutativity (C) laws for $\times$ to achieve the same effect as with the distributivity laws for the general contraction rule.

Since E-unification up to the theory (A)+(C)+(U) is known to be NP-complete [8], we obtain the following hardness result.

Proposition 4. *Type inference for the variant of* IntML *obtained by replacing (*CONTR*) with (*CONTR2*) and by letting $\cong$ be defined by (A)+(C)+(U) is NP-hard.*

We do not know if type inference for this variant of IntML is also in NP; it is unclear how to reduce $\precsim$-constraints to $\cong$-constraints in general.

We side-step the difficulty of reducing $\precsim$ to $\cong$ by noticing that the relation $\precsim$ is larger than it needs to be in order to obtain the properties discussed in Sec. 2, which were the original reasons for introducing $\precsim$ into the type system. In the IntML-variant with (CONTR2), it would suffice to be able to treat index types of the form $A_1 \times \cdots \times A_n$ as if they were multisets $\{A_1, \ldots, A_n\}$. The relation $\precsim$ does more than that.

For any working class type A define the multiset $M(A)$ represented by it inductively as follows: $M(1) = \emptyset$, $M(\alpha) = \{\alpha\}$, $M(A + B) = \{A + B\}$ and $M(A \times B) = M(A) \cup M(B)$.

We define $\precsim_M$ such that $A \precsim_M B$ holds if and only if $M(A)$ is a sub-multiset of $M(B)$ in the sense that there exists a multiset E with $M(B) = M(A) \cup E$. Syntactic congruence is then defined by $A \cong_M B$ if and only if $M(A) = M(B)$.

With this definition of $\precsim_M$ and $\cong_M$, the variant of IntML with $\precsim_M$ instead of $\precsim$ is not only well-behaved, its type inference constraints can also easily be reduced to E-unifcation up to (A)+(C)+(U). First, any constraint $A \precsim_M B$ may equivalently be replaced by $A \times \alpha \cong_M B$ for some fresh variable α. Then, solving $A \cong_M B$ amounts to unifying A and B up to (A)+(C)+(U), where each type of the form $C + D$ is considered a constant. Thus, type inference is equally hard as unification of terms over $(\times, 1)$ up to (A)+(C)+(U) and infinitely many constants, an NP-complete problem [3]:

Proposition 5. *Type inference for the variant of* IntML *obtained by replacing (*CONTR*) with (*CONTR2*) and by using $\precsim_M$ in rule (*STRUCT*) is NP-complete.*

Quick and Simple Constraint Solving. For use in practice, an NP-complete algorithm for type inference may still be too complex, with regard to running time and to implementation effort. Here we present a quick and simple algorithm for type inference that we have found to be practically useful (it handles all the applications we know), even though it does not handle the full the type system or find most general types.

For this algorithm we take the congruence $\cong$ to be syntactic equality, i.e. we consider ordinary unification instead of E-unification. In this case the relations $\leq$ and $\precsim$ coincide, so we will write $A \leq B$ for $A \precsim B$ in the rest of this section.

The restriction to syntactic equality alone is not enough to make type inference easy. We must also restrict the context Γ in the type inference problem $\mathcal{T}(\Sigma \mid \Gamma, t)$. We will consider here contexts that do not impose any constraints on index types. Such a restriction not only reflects the typical use of type inference in practice, where one does not have information about index types and would like to have them inferred. It also makes type inference significantly easier. If we do not impose such a restriction then type inference remains NP-hard, even when $\cong$ is syntactic equality.

Proposition 6. *Even if we let the congruence relation $\cong$ be syntactic equality, it is NP-hard to decide if a constraint set returned by the type inference function has a solution.*

We now identify a class of type inference problems $\mathcal{T}(\Sigma \mid \Gamma, t)$ that are useful in practice and that can be solved easily. We obtain this class by restricting Γ to be an *unconstrained context*, which does not impose restrictions on index types, and by restricting t to contain only instances of $\mathsf{hack}_X(c.f)$ where the index types in positive positions in X are all type variables.

We next formulate the restrictions precisely. To allow for a compact formulation of freshness assumptions, we choose a partition of the set of type variables into two (arbitrary) disjoint infinite subsets Var and $IdxVar$.

A *positively (resp. negatively) unconstrained type* is an upper class type in which all index types in positive (resp. negative) position are variables from $IdxVar$, and variables from $IdxVar$ may only appear as index types. Formally, the positively and negatively unconstrained types are defined by the grammar below, in which A ranges over working class types with variables in Var, α ranges over Var and β over $IdxVar$.

$$\begin{array}{rr} \text{positively unconstrained:} & P ::= \alpha \mid [A] \mid P \otimes P \mid A \cdot N \multimap P \\ \text{negatively unconstrained:} & N ::= \alpha \mid [A] \mid N \otimes N \mid \beta \cdot P \multimap N \end{array}$$

An *unconstrained type* is a type that is both positively and negatively unconstrained, i.e. in which all index types are variables from $IdxVar$.

A *unconstrained context* is a context of the form $x_1 : \alpha_1 \cdot X_1, \ldots, x_n : \alpha_n \cdot X_n$, where all α_i are variables from $IdxVar$ and all X_i are unconstrained types.

We can now formulate conditions that make type inference easy.

Proposition 7. *Let $\cong$ be syntactic equality. Let Σ be a working class context containing only type variables from Var, let Γ be an unconstrained context, and let t be a term that may contain $\mathsf{hack}_X(c.f)$ as a subterm only if X is a positively unconstrained type. For such Σ, Γ and t we can compute in polynomial time either a type X and a substitution σ with $\Sigma\sigma \mid \Gamma\sigma \vdash t\sigma : X\sigma$; or reject if no such X and σ exist.*

We believe that this proposition captures a practically useful class of type inference problems. It contains all the practical examples we know, including the graph algorithm example from the previous section. Restricting to unconstrained contexts seems reasonable, since the index types capture information about the space usage of programs but are not important for understanding the meaning of programs. One would like to not worry about the index types and leave it to the type inference to fill them in.

For the proof of Prop. 7 we may use the following simple algorithm:

1. Compute $\mathcal{T}(\Sigma \mid \Gamma, t)$, which gives a pair (X, C).
2. Compute the most general unifier σ of all equations in C (ignoring the inequations) and set $I = \{A\sigma \leq B\sigma \mid (A \leq B) \in C\}$. Reject if the equations are not unifiable.
3. While possible, choose from I two inequalities $A \leq C$ and $B \leq C$ with the same upper bound C and replace them with $A + B \leq C$.
4. Compute the most general unifier τ of the set $\{A = B \mid (A \leq B) \in I\}$.
5. Return X and $\sigma; \tau$ as the final result.

We cannot hope for this algorithm to compute most general solutions, simply because the constraint set C may be infinitary. This means that in general there does not exist a finite set S of substitutions that solves a given constraint set C in the sense that any solution of C is an instance of one of the substitutions in S.

Proposition 8. *If we let the congruence relation $\cong$ be syntactic equality, then the constraint solving problem is infinitary.*

Although the simple algorithm above does not in general find most general types, practical experiments indicate that it finds useful typings. In our experience with a prototype implementation, the fact that the algorithm does not compute most general types means that the index types are a little larger than they need to be. The effect of this is an increased space usage of IntML-programs, since the index types appear in the types of messages that are being stored during the computation. In examples like those from Sec. 3, the space usage nevertheless quite reasonable.

5 Conclusion

The usefulness of a type system for a programming language depends on how well it strikes a balances between the benefits gained by types and the overhead of dealing with types when writing programs. Here we have shown that the overhead of the IntML type system can be reduced substantially by means of type inference. With its index types, the IntML type system is precise enough to guarantee space bounds on IntML-programs, and so it is not a surprise that full type inference turns out to be quite hard. Nevertheless, we were able to identify a class of type inference instances that can be solved quickly and easily and that still appears to be useful in practice. To substantiate this claim, we have expressed a typical LOGSPACE-graph algorithm in IntML. With type inference it could be programmed with surprisingly little overhead due to the type system.

References

1. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full Abstraction for PCF. Inf. Comput. **163**(2) (2000) 409–470
2. Atassi, V., Baillot, P., Terui, K.: Verification of ptime reducibility for system F terms: Type inference in dual light affine logic. Logical Methods in Computer Science **3**(4) (2007)
3. Baader, F., Snyder, W.: Unification theory. In Robinson, J.A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier and MIT Press (2001) 445–532
4. Burrell, M.J., Cockett, R., Redmond, B.F.: Pola: a language for PTIME programming. In: Workshop on Logic and Computational Complexity (LCC). (2009)
5. Cook, S., McKenzie, P.: Problems complete for deterministic logarithmic space. Journal of Algorithms **8**(3) (1987) 385–394
6. Dal Lago, U., Schöpp, U.: Functional programming in sublinear space. In Gordon, A.D., ed.: ESOP. Volume 6012 of Lecture Notes in Computer Science., Springer (2010) 205–225
7. Hermann, M., Kolaitis, P.G.: Computational complexity of simultaneous elementary matching problems. J. Autom. Reasoning **23**(2) (1999) 107–136
8. Kapur, D., Narendran, P.: Complexity of unification problems with associative-commutative operators. J. Autom. Reasoning **9**(2) (1992) 261–288
9. Pottier, F., Rémy, D.: The essence of ML type inference. In Pierce, B.C., ed.: Advanced Topics in Types and Programming Languages. MIT Press (2005) 389–489
10. Tidén, E., Arnborg, S.: Unification problems with one-sided distributivity. J. Symb. Comput. **3**(1/2) (1987) 183–202

Appendix

A Proofs for Section 2

Lemma 2. *If $A \precsim B$ and $B \precsim A$ then $A \cong B$.*

Proof. By definition we have $A \leq B' \cong B$ and $B \leq A' \cong A$. Consider the map e from types to positive natural numbers defined by $e(1) = 1$, $e(\alpha) = 2$, $e(A + B) = e(A) + e(B)$ and $e(A \times B) = e(A) \cdot e(B)$. It follows easily that $A \cong B$ implies $e(A) = e(B)$ and that $A \leq B$ implies $e(A) \leq e(B)$. Therefore, we have $e(A) \leq e(B') = e(B)$ and $e(B) \leq e(A') = e(A)$. But together, these two facts imply $e(A) = e(B')$ and $e(B) = e(A')$. The only basic cases in the definition of $\leq$ that preserve e are $C \leq 1 \times C$ and $C \leq C \times 1$. But then B' must be obtained from A by a series of replacements of some C by $C \times 1$ or $1 \times C$. In other words, $A \cong B'$. Analogously we obtain $B \cong A'$, so that the result follows by transitivity of $\cong$. □

B Definitions from Section 3

Omitted definition of $\mathit{nextEdge}_\alpha$:

$$
\begin{array}{l}
\mathit{nextEdge}_\alpha : \mathit{Graph}(\alpha) \to [\alpha \times \alpha] \to [\alpha \times \alpha] \\
\quad \lambda \mathit{graph}.\, \lambda \mathit{inputedge}. \\
\quad \mathsf{let}\ \mathit{inputedge}\ \mathsf{be}\ [e]\ \mathsf{in} \\
\quad (\mathtt{loop}\ (\lambda e'.\, \mathsf{let}\ e'\ \mathsf{be}\ [c]\ \mathsf{in} \\
\qquad\qquad \mathit{if}\ (\mathit{edge}\ \mathit{graph}\ [c]) \\
\qquad\qquad\quad [\mathsf{return}(c)] \\
\qquad\qquad\quad [\mathsf{continue}(\langle \mathit{src}\ c, \mathit{cysucc}_\alpha\ (\mathit{dst}\ c)\rangle)]) \\
\qquad [\langle \mathit{src}\ e, \mathit{cysucc}_\alpha\ (\mathit{dst}\ e)\rangle]
\end{array}
$$

C Proofs for Section 4

We outline the proof of completeness. For reference, we include the full set of rules for upper class type inference in Fig. 8.

Proposition 2 (Completeness). *The following are true.*

1. *If $\Sigma\sigma \vdash f\sigma : A\sigma$ then there exist B and C with $\mathcal{T}(\Sigma, f) = (B, C)$ such that σ can be extended to a type substitution σ' satisfying $A\sigma = B\sigma'$ and $\sigma' \models C$.*
2. *If $\Sigma\sigma \mid \Gamma\sigma \vdash t\sigma : X\sigma$ then there exist Y and C with $\mathcal{T}(\Sigma \mid \Gamma, t) = (Y, C)$ such that σ can be extended to a type substitution σ' satisfying $X\sigma = Y\sigma'$ and $\sigma' \models C$.*

For the proof of this proposition, we need a strengthening lemma, which is proved by straightforward induction on derivations.

$$\text{(VAR)}\ \frac{\Gamma \text{ contains } x : A{\cdot}X}{\mathcal{T}(\Sigma \mid \Gamma, x) = (X, \{1 \precsim A\})} \qquad \text{(}{\multimap}\text{I)}\ \frac{\mathcal{T}(\Sigma \mid (\Gamma,\ x : \alpha{\cdot}\beta), s) = (X, C) \qquad \alpha, \beta \text{ fresh}}{\mathcal{T}(\Sigma \mid \Gamma, \lambda x.\ s) = (\alpha \cdot \beta \multimap X, C)}$$

$$\text{(}{\multimap}\text{E)}\ \frac{\begin{array}{ll}\mathcal{T}(\Sigma \mid \Gamma|_s, s) = (Y_1, C_1) & \\ \Gamma|_t = (x_1 : A_1{\cdot}X_1, \ldots, x_n : A_n{\cdot}X_n) & \vec{\alpha}, \beta, \gamma, \delta \text{ fresh} \\ \mathcal{T}(\Sigma \mid (x_1 : \alpha_1{\cdot}X_1, \ldots, x_n : \alpha_n{\cdot}X_n), t) = (Y_2, C_2) & FV(s) \cap FV(t) = \emptyset\end{array}}{\mathcal{T}(\Sigma \mid \Gamma, s\ t) = (\beta, C_1 \cup C_2 \cup \{Y_1 = \delta \cdot \gamma \multimap \beta, Y_2 = \gamma\} \cup \{\delta \times \alpha_i \precsim A_i \mid 1 \le i \le n\}))}$$

$$\text{(}{\otimes}\text{I)}\ \frac{\begin{array}{ll}\mathcal{T}(\Sigma \mid \Gamma|_s, s) = (Y_1, C_1) & \\ \mathcal{T}(\Sigma \mid \Gamma|_t, t) = (Y_2, C_2) & FV(s) \cap FV(t) = \emptyset\end{array}}{\mathcal{T}(\Sigma \mid \Gamma, \langle s, t\rangle) = (Y_1 \otimes Y_2, C_1 \cup C_2)}$$

$$\text{(}{\otimes}\text{E)}\ \frac{\begin{array}{ll}\Gamma|_s = (x_1 : A_1{\cdot}X_1, \ldots, x_n : A_n{\cdot}X_n) & \\ \mathcal{T}(\Sigma \mid (x_1 : \alpha_1{\cdot}X_1, \ldots, x_n : \alpha_n{\cdot}X_n), s) = (Y_1, C_1) & \vec{\alpha}, \beta_1, \beta_2, \gamma \text{ fresh} \\ \mathcal{T}(\Sigma \mid (\Gamma|_t,\ x : \gamma{\cdot}\beta_1,\ y : \gamma{\cdot}\beta_2), t) = (Y_2, C_2) & FV(s) \cap FV(t) = \emptyset\end{array}}{\mathcal{T}(\Sigma \mid \Gamma, \mathsf{let}\ s\ \mathsf{be}\ \langle x, y\rangle\ \mathsf{in}\ t) = (Y_2, C_1 \cup C_2 \cup \{Y_1 = \beta_1 \otimes \beta_2\} \cup \{\gamma \times \alpha_i \precsim A_i \mid 1 \le i \le n\})}$$

$$\text{(CONTR)}\ \frac{\begin{array}{ll}\Gamma|_s = (x_1 : A_1{\cdot}X_1, \ldots, x_n : A_n{\cdot}X_n) & \\ \mathcal{T}(\Sigma \mid (x_1 : \alpha_1{\cdot}X_1, \ldots, x_n : \alpha_n{\cdot}X_n), s) = (Y_1, C_1) & \vec{\alpha}, \beta, \gamma, \delta \text{ fresh} \\ \mathcal{T}(\Sigma \mid (\Gamma|_t,\ x : \delta{\cdot}\gamma,\ x : \beta{\cdot}\gamma), t) = (Y_2, C_2) & FV(s) \cap FV(t) = \emptyset\end{array}}{\mathcal{T}(\Sigma \mid \Gamma, \mathsf{copy}\ s\ \mathsf{as}\ x, y\ \mathsf{in}\ t) = (Y_2, C_1 \cup C_2 \cup \{Y_1 = \gamma\} \cup \{(\delta + \beta) \times \alpha_i \precsim A_i \mid 1 \le i \le n\})}$$

$$\text{(CASE)}\ \frac{\begin{array}{ccc}\multicolumn{3}{c}{\mathcal{T}((\Sigma,\ c{:}\alpha) \mid \Gamma, s) = (Y_1, C_1)} \\ \mathcal{T}(\Sigma, f) = (A, C) & \mathcal{T}((\Sigma,\ d{:}\beta) \mid \Gamma, t) = (Y_2, C_2) & \alpha, \beta \text{ fresh}\end{array}}{\mathcal{T}(\Sigma \mid \Gamma, \mathsf{case}\ f\ \mathsf{of}\ \mathsf{inl}(c) \Rightarrow s \mid \mathsf{inl}(d) \Rightarrow t) = (Y_1, C \cup C_1 \cup C_2 \cup \{A = \alpha + \beta, Y_1 = Y_2\})}$$

$$\text{([ ]I)}\ \frac{\mathcal{T}(\Sigma, f) = (A, C)}{\mathcal{T}(\Sigma \mid \Gamma, [f]) = ([A], C)}$$

$$\text{([ ]E)}\ \frac{\begin{array}{ll}\mathcal{T}(\Sigma \mid \Gamma|_s, s) = (Y_1, C_1) & \\ \Gamma|_t = (x_1 : A_1{\cdot}X_1, \ldots, x_n : A_n{\cdot}X_n) & \vec{\alpha}, \beta \text{ fresh} \\ \mathcal{T}((\Sigma,\ c{:}\beta) \mid (x_1 : \alpha_1{\cdot}X_1, \ldots, x_n : \alpha_n{\cdot}X_n), t) = (Y_2, C_2) & FV(s) \cap FV(t) = \emptyset\end{array}}{\mathcal{T}(\Sigma \mid \Gamma, \mathsf{let}\ s\ \mathsf{be}\ [c]\ \mathsf{in}\ t) = (Y_2, C_1 \cup C_2 \cup \{Y_1 = [\beta]\} \cup \{\beta \times \alpha_i \precsim A_i \mid 1 \le i \le n\})}$$

$$\text{(HACK)}\ \frac{\mathcal{T}((\Sigma,\ c{:}X^-), f) = (A, C) \qquad p(X) = Y}{\mathcal{T}(\Sigma \mid \Gamma, \mathsf{hack}_X(c.f)) = (p(X), C \cup \{A = X^+\} \cup c(X, Y))}$$

Here, $p(X)$ is the function that replaces each index type in X that is not already a variable in $IdxVar$ with a fresh variable from $IdxVar$. Formally:

$$\begin{aligned} p([A]) &= [A] \\ p(X_1 \otimes X_2) &= p(X_1) \otimes p(X_2) \\ p(\alpha \cdot X \multimap Y) &= \alpha \cdot p(X) \multimap p(Y) \ \text{ if } \alpha \in IdxVar \\ p(A \cdot X \multimap Y) &= \alpha \cdot p(X) \multimap p(Y) \ \text{ otherwise, with } \alpha \in IdxVar \text{ fresh} \end{aligned}$$

The function $c(X, Y)$ returns the set of constraints needed to obtain $(X \precsim Y)$, where $Y = p(X)$.

$$\begin{aligned} c([A], [A]) &= \emptyset \\ c(X_1 \otimes X_2, X_1' \otimes X_2') &= c(X_1, X_1') \cup c(X_2, X_2') \\ c(\alpha \cdot X \multimap Y, \alpha \cdot X' \multimap Y') &= c(X', X) \cup c(Y, Y') \ \text{ if } \alpha \in IdxVar \\ c(A \cdot X \multimap Y, A' \cdot X' \multimap Y') &= \{A \le A'\} \cup c(X', X) \cup c(Y, Y') \text{ otherwise} \end{aligned}$$

Fig. 8. Upper Class Type Inference

Lemma 3 (Strengthening). *If* $\Sigma \mid \Gamma, x : A{\cdot}X, \Delta \vdash s : Y$ *and* $x \notin FV(s)$ *then* $\Sigma \mid \Gamma, \Delta \vdash s : Y$.

Proof (Proposition 2 (Sketch)). The proof goes by induction on the terms t and f. We outline the representative case where t is an application $t_1\ t_2$. In this case we can assume that the derivation of $\Sigma\sigma \mid \Gamma\sigma \vdash t\sigma : X\sigma$ has the following form.

$$\dfrac{\dfrac{\dfrac{\Pi_1 \qquad\qquad \Pi_2}{\Sigma\sigma \mid (\Gamma|_{t_1})\sigma \vdash t_1\sigma : A \cdot Z \multimap (X\sigma) \qquad \Sigma\sigma \mid (\Gamma|_{t_2})\sigma \vdash t_2\sigma : Z}}{(\textsc{Struct})\ \dfrac{\Sigma\sigma \mid (\Gamma|_{t_1}, A \cdot (\Gamma|_{t_2}))\sigma \vdash (t_1\ t_2)\sigma : X\sigma}{\Sigma\sigma \mid (\Gamma|_{t_1}, \Delta)\sigma \vdash (t_1\ t_2)\sigma : X\sigma}}}{(\textsc{Weak}, \textsc{Exch})\ \dfrac{\Sigma\sigma \mid (\Gamma|_{t_1}, \Delta)\sigma \vdash (t_1\ t_2)\sigma : X\sigma}{\Sigma\sigma \mid \Gamma\sigma \vdash (t_1\ t_2)\sigma : X\sigma}}$$

This can be seen using the strengthening lemma and by observing that any other use of (STRUCT) could either be permuted with the structural rules at the end of the derivation or pushed into Π_1.

Let $(x_1 : A_1{\cdot}X_1, \ldots, x_n : A_n{\cdot}X_n) := \Gamma|_{t_1}$. Choose fresh type variables $\alpha_1, \ldots, \alpha_n, \beta, \gamma$ and δ and let σ' be the substitution $\sigma_1[\alpha_1 \mapsto A_1] \ldots [\alpha_n \mapsto A_n][\beta \mapsto X\sigma][\gamma \mapsto Z][\delta \mapsto A]$. Define Γ' to be $x_1 : \alpha_1{\cdot}X_1, \ldots, x_n : \alpha_n{\cdot}X_n$.

With these definitions, the conclusion of the above derivation then is $\Sigma\sigma' \mid \Gamma\sigma' \vdash (t_1\ t_2)\sigma' : \beta\sigma'$. The conclusions of Π_1 and Π_2 are $\Sigma\sigma' \mid (\Gamma|_{t_1})\sigma' \vdash t_1\sigma' : (\delta{\cdot}\gamma \multimap \beta)\sigma'$ and $\Sigma\sigma' \mid \Gamma'\sigma' \vdash t_2\sigma' : \gamma\sigma'$ respectively.

We apply the induction hypothesis to the two leaves to obtain first $\mathcal{T}(\Sigma \mid \Gamma|_{t_1}, t_1) = (Z_1, C_1)$ and that σ' can be extended to some σ_1' with $Z_1\sigma_1' = (\delta \cdot \gamma \multimap \beta)\sigma'$ and with $\sigma_1' \models C_1$; and second $\mathcal{T}(\Sigma \mid \Gamma', t_2) = (Z_1, C_2)$ and that σ' can be extended to some σ_2' with $Z_2\sigma'' = \gamma\sigma'$ and with $\sigma_2' \models C_2$.

Since the type inference rule in Fig. 7 are closed under permutation of type variables, as is easily seen, we can rename the derivations of $\mathcal{T}(\Sigma \mid \Gamma|_{t_1}, t_1) = (Z_1, C_1)$ and $\mathcal{T}(\Sigma \mid \Gamma', t_2) = (Z_1, C_2)$ so that none of the freshly chosen variables is used in both derivations. Hence, σ_1' and σ_2' can in fact be extended to a single substitution σ''. This substitution then validates $C_1 \cup C_2$.

From the induction hypothesis, we get $\sigma'' \models Z_1 = (\delta \cdot \gamma \multimap \beta)$. and $\sigma'' \models Z_2 = \gamma$.

We also have $\sigma'' \models \delta \cdot \alpha_i \precsim A_i$ for $1 \leq i \leq n$ by the shape of the derivation.

But by the type inference rule for application we also have $\mathcal{T}(\Sigma \mid \Gamma, t_1\ t_2) = (\beta, C_1 \cup C_2 \cup \{Z_1 = \delta \cdot \gamma \multimap \beta, Z_2 = \gamma\} \cup \{\delta \cdot \alpha_i \precsim A_i \mid 1 \leq i \leq n\})$. We have thus shown that σ can be extended to a substitution σ'' that satisfies all the constraints as well as $\beta\sigma'' = X\sigma$. This is what we were required to show. □

D Proofs for Section 4.2

Proposition 3. *Solving the constraint sets that arise in* IntML *type inference is at least as hard as equational unification for (U)+(D).*

Proof. First we show that essentially any set of $\cong$-constraints can appear in type inference.

Let $\{A_1 \cong B_1, \ldots, A_n \cong B_n\}$ be any set of congruences. We construct arguments for the type inference function such that the returned constraints is equivalent to this set of congruences.

Let X be any variable free upper class type, e.g. $X = [1]$. Observe that we have $\mathcal{T}(\cdot \mid (f : 1{\cdot}(C \cdot X \multimap X), x : D{\cdot}X), f\ x) =$
$(\beta, \{C \cdot X \multimap X = \delta \cdot \gamma \multimap \beta,\ X = \gamma,\ \delta \times \alpha \precsim D,\ 1 \precsim 1, 1 \precsim \alpha\})$ where all type variables are fresh. The equalities in the constraint set force $\delta = C$ and $\gamma = \beta = X$. Hence, the constraint set is equivalent to $\{C \times \alpha \precsim D,\ 1 \precsim \alpha\}$.

If we let Γ be the context

$$f_1 : 1{\cdot}(C_1 \cdot X \multimap X),\ x_1 : D_1{\cdot}X,\\ \ldots, f_m : 1{\cdot}(C_m \cdot X \multimap X),\ x_m : D_m{\cdot}X,$$

then the constraint set arising from

$$\mathcal{T}(\cdot \mid \Gamma, \langle f_1\ x_1, \langle \cdots \langle f_{m-1}\ x_{m-1}, f_m\ x_m \rangle \cdots \rangle)$$

can similarly seen to be equivalent to

$$\{C_i \times \alpha_i \precsim D_i,\ 1 \precsim \alpha_i \mid 1 \leq i \leq m\}.$$

Now we notice that $\{C \times \alpha \precsim D,\ D \times \beta \precsim C\}$ is equivalent to $\{C \cong D\}$. To this end notice that $C \times \alpha \precsim D$ implies $(C \times \alpha) \times \beta \precsim D \times \beta$, which with the other structural inequality gives $(C \times \alpha) \times \beta \precsim C$. Likewise we get $(D \times \beta) \times \alpha \precsim D$. Now we observe that these two inequalities can only be solved if α and β are both assigned to 1. Consider any solution σ of the constraints. Consider again the map e from variable-free types to positive natural numbers defined by $e(1) = 1$, $e(A + B) = e(A) + e(B)$ and $e(A \times B) = e(A) \cdot e(B)$ and extend it to types with free variables by $e(\alpha) = 2$ for all α. It is evident that $A \leq B$ implies $e(A) \leq e(B)$. Since σ is a solution we thus get $e(D\sigma) \cdot e(\sigma(\beta)) \cdot e(\sigma(\alpha)) \leq e(D\sigma)$. It follows that $e(\sigma(\beta))$ and $e(\sigma(\alpha))$ must both be 1. But the only type C with $e(C) = 1$ is $C = 1$. Hence $\sigma(\beta) = \sigma(\alpha) = 1$. The constraint set $\{C \times \alpha \precsim D,\ D \times \beta \precsim C\}$ is therefore equivalent to $\{C \times 1 \precsim D,\ D \times 1 \precsim C\}$, which itself is equivalent to $\{C \precsim D,\ D \precsim C\}$, which finally is equivalent to $\{C \cong D\}$.

By choosing $m = 2n$ and $C_1 = A_1$, $D_1 = B_1$, $C_2 = B_1$, $D_2 = A_1$, etc. we can therefore construct arguments to the type inference function that yield a constraint set equivalent to the one we set out to capture.

Thus, for any set of (U)+(D)-equations, we can effectively construct a term that gives rise to an equivalent constraint set. Therefore, if we can solve constraint sets that arise in type inference, then we can also solve the set of (U)+(D)-equations. □

Proposition 4. *Type inference for the variant of* IntML *obtained by replacing (*Contr*) with (*Contr2*) and by letting $\cong$ be defined by (A)+(C)+(U) is NP-hard.*

Proof. We argue as in the proof of Prop. 3 that any given set of $\cong$-constraints can actually arise in type inference. Then the result follows by NP-hardness of unification up to (A)+(C)+(U). □

Proposition 5. *Type inference for the variant of* IntML *obtained by replacing (*Contr*) with (*Contr2*) and by using $\precsim_M$ in rule (*Struct*) is NP-complete.*

Proof. First we note that the constraint sets arising in type inference have size linear in the type inference problem. To solve such constraint sets we note that solving $A \cong_M B$ amounts to unifying A and B up to (A)+(C)+(U), where we consider subterms of the form $C + D$ as constant symbols. Also, $A \precsim_M B$ is equivalent to $A \times \alpha \cong_M B$, given that α is fresh. Thus, the constraint sets arising in type inference can be solved by unification up to (A)+(C)+(U), which is in NP. Combining these observations, we obtain an NP-algorithm for type inference.

For NP-hardness we argue again as in the proof of Prop. 3 that any given set of $\cong$-constraints can actually arise in type inference. We note that $A \cong_M B$ implies that A and B are equal up to (A)+(C)+(U), where again we consider subterms of the form $C + D$ as constant symbols. Hence it is sufficient to show that (A)+(C)+(U)-unification is NP-hard for sets of equations over a signature that contains infinitely many constant symbols (here types of the form $C + D$) in addition to $\times$ and 1. But Hermann & Kolatitis [7] show that matching up to (A)+(C)+(U) is already NP-hard if there is a single constant in addition to $\times$ and 1. Hence we obtain NP-hardness of type checking.

E Proofs for Section 4.2

Proposition 6. *Even if we let the congruence relation $\cong$ be syntactic equality, it is NP-hard to decide if a constraint set returned by the upper class type inference function has a solution.*

Proof. We prove NP-hardness by reducing from SAT. Given any CNF φ, we build a constraint set as follows: For any variable x in φ we choose two type variables α_x^t and α_x^f. For any variable x we add the constraint $\alpha_x^t + \alpha_x^f \leq (1+(1+1)) \times ((1+1)+1)$. For a literal l we define the type $A(l) = \alpha_x^t + \alpha_x^f$ if $l = x$ and $A(l) = \alpha_x^f + \alpha_x^t$ if $l = \neg x$. Now, for any clause $C = \{l_1, \ldots, l_n\}$ in φ, we add the constraint $1 + (1+1) \leq A(l_1) \times A(l_2) \times \cdots \times A(l_n)$.

The constraints enforce that for each x there are three possibilities for (α_x^t, α_x^f): $(1, 1+1)$, $(1+1, 1)$ or $(1, 1)$. The first case stands for x being true and the second for x being false. The last case means that the value of x does not matter. The constraint for a clause forces at least one literal to be satisfied. Now, if we have a satisfying assignment for φ, then by choosing the α_x-variables accordingly we can satisfy the constraints. Conversely, if all the constraints are satisfied then we can read off a satisfying assignment for φ. This shows NP-hardness of constraint solving when $\cong$ is syntactic equality.

It remains to show that these hard instances can really appear in type inference. We notice: any constraint $A \leq B$ is equivalent to $A \times \alpha \leq B \times 1$ where α is a fresh variable. That any solution to $A \leq B$ gives one of $A \times \alpha \leq B \times 1$ is obvious; just set α to 1. In the other direction there are two possible cases. A solution either validates $A \leq B$ and $\alpha \leq 1$; or it validates $A \times \alpha \leq B' \leq B' \times 1 \leq B \times 1$ and $B' \leq B$ for some B'. In the first case we are done. In the second case we get $A \leq A \times \alpha \leq B' \leq B$, so we are also done.

But in the proof of Proposition 3 we have already observed that any set of constraints of the form $A \times \alpha \leq B \times 1$ can arise from type inference. Hence, we can construct an instance of type inference that generates constraints equivalent to the ones used for the NP-hardness argument above. □

Proposition 7. *Assume that the congruence relation $\cong$ is syntactic equality. Let Σ be a working class context containing only type variables from Var, let Γ be an unconstrained context, and let t be a term that may contain $\mathsf{hack}_X(c.f)$ as a subterm only if X is a positively unconstrained type. For such Σ, Γ and t we can compute in polynomial time either a type X and a substitution σ, such that $\Sigma\sigma \mid \Gamma\sigma \vdash t\sigma \colon X\sigma$ is derivable; or reject if no such substitution exists.*

To prove this proposition, we show that the algorithm from Section 4.2 is sound and complete, provided Σ, Γ and t satisfy the assumption in the proposition. It follows from soundness of type inference (Prop. 1) that the algorithm outputs a correct solution, if it gets to the last step. It remains to show that it does not reject inputs that can be solved. To prove this, we need a few results about the form the constraint set C obtained in step 1 can have.

A *unconstrained substitution* is a substitution with the following properties:

- For each $\beta \in Var$, $\sigma(\beta)$ is an unconstrained type.
- For each $\alpha \in IdxVar$, $\sigma(\alpha)$ is a type variable in $IdxVar$.

An *unconstrained solution* for a set of constraints C is an unconstrained substitution σ, which is a solution for C.

Lemma 4. *Unconstrained substitutions are closed under composition.*

Lemma 5. *Let C_1 and C_2 be finite sets of equations such that $C_1 \cup C_2$ is solvable. Assume that any most general unifier of C_1 and C_2 is an unconstrained substitution. Then there exist most general unifiers σ_1, σ_2 and σ of C_1, C_2 and $C_1 \cup C_2$ respectively, such that $\sigma = \sigma_1; \tau$ and $\sigma = \sigma_2; \tau$ hold for some unconstrained substitution τ.*

Proof. Let σ_1 and σ_2 be most general solutions for C_1 and C_2 and assume that they are unconstrained. Assume moreover that each term $\sigma_1(\alpha)$ contains only fresh variables not appearing in C_1, C_2 of any $\sigma_2(\beta)$, which can always be arranged by variable renaming.

Now notice that a most general unifier of $C_1 \cup C_2$ (which exists since we have assumed this set to be solvable) may be written as $\sigma_1; \tau = \sigma_2; \tau$, where τ is the most general unifier of the set $\{\sigma_1(\beta) = \sigma_2(\beta) \mid \beta \in Var \cup IdxVar\}$ (this uses the freshness assumption). Since both σ_1 and σ_2 are unconstrained solutions, it follows that these equations may only force $\tau(\beta)$ to be an unconstrained type if $\beta \in Var$ or to be a variable in $IdxVar$. In other words, τ is an unconstrained solution. □

Lemma 6. *Let Σ be a working class context containing only type variables from Var, let Γ be an unconstrained context, and let t be a term that contains only subterms of the from $\mathsf{hack}_X(c.f)$ in which X is a positively unconstrained type.*

Assume $\mathcal{T}(\Sigma \mid \Gamma, t) = (X, C)$ and that σ is a most general unifier of all the equations in C (ignoring the inequations). Then the following are true.

1. *$X\sigma$ is an unconstrained type.*
2. *σ is an unconstrained solution for C.*
3. *The constraint set C contains only inequalities of the form $A \leq \alpha$ where $\alpha \in IdxVar$, $FV(A) \subseteq IdxVar$ and A is not a variable in $IdxVar$.*

Proof. First we observe that by renaming the freshly chosen variables, we can assume that all upper class contexts in the derivation of $\mathcal{T}(\Sigma \mid \Gamma, t) = (X, C)$ are unconstrained.

With this assumption the proof then goes by induction on the derivation of $\mathcal{T}(\Sigma \mid \Gamma, t) = (X, C)$. We consider representative cases:

- ($\multimap$-I) In this case, points 2 and 3 follow immediately from the induction hypothesis, since the constraint sets in premise and conclusion of this rule are the same. Point 1 follows because by point 1 of the induction hypothesis $X\sigma$ is an unconstrained type. Since, by point 2 of the induction hypothesis, the substitution σ is unconstrained, the type $(\alpha \cdot \beta \multimap X)\sigma$ must also be unconstrained, which shows point 1.
- ($\multimap$-E) Let σ be a most general unifier of the equations in C.
  By Lemma 5 there are most general unifiers σ_1, σ_2 and σ' of C_1, C_2 and $C_1 \cup C_2$ respectively, such that we have $\sigma' = \sigma_1; \tau$ and $\sigma' = \sigma_2; \tau$ for some unconstrained τ. We can apply the induction hypothesis to obtain points 1–3 for Y_1 and σ_1 as well as for Y_2 and σ_2. As unconstrained substitutions are closed under composition, we obtain moreover that σ' is unconstrained.
  A most general unifier of the equations in C can be obtained obtained by unifying the equations the the new equations are $Y_1\sigma' = (\delta \cdot \gamma \multimap \beta)\sigma'$ and $Y_2\sigma' = \gamma\sigma'$. Since $Y_1\sigma_1$ and $Y_2\sigma_2$ are unconstrained types by induction hypothesis and since τ is unconstrained, because substitution with an unconstrained substitution in an unconstrained type gives an unconstrained type, it follows that $Y_1\sigma' = Y_1\sigma_1\tau$ and $Y_2\sigma' = Y_2\sigma_2\tau$ are also unconstrained types.
  With this observation it follows as in Lemma 5 that after solving the new equations, we still have an unconstrained substitution σ and $\beta\sigma$ must be unconstrained.
  Finally, point 3 follows immediately from the induction hypothesis and by examination of the new inequalities.

□

Proof (Proposition 7). By soundness and completeness of type inference, it suffices to show that the algorithm solves the constraint set C obtained in point 1 if and only if this set has a solution.

To show this, we note first that C is solvable if and only if the equations in it can be solved and the set of inequalities I has a solution. Here we show that, due to the unconstrained nature of the type inference problem, the set I always has a solution if the equations in C can be solved and that such a solution can be computed using steps 3 and 4.

From Lemma 6.3 it follows that the set I contains only inequations $A \leq \alpha$ with a variable from $IdxVar$ as an upper bound.

We notice that we can moreover order the inequations I as

$$A_1 \leq \alpha_1, \ldots, A_n \leq \alpha_n$$

such that we have $\alpha_j \notin FV(A_i)$ for all $1 \leq i \leq j \leq n$. Suppose this were not the case. Then there would exist indices $i_1, \ldots, i_k$ with $\alpha_{i_l} \in FV(A_{i_{l+1 \bmod k}})$ for $1 \leq l \leq k$. To keep the notation simple, let us do just the case $k = 2$; the other cases

follow by the same argument. Thus assume $A \leq \alpha$ and $B \leq \beta$ in I with $\beta \in FV(A)$ and $\alpha \in FV(B)$. But then we would also get $B[A/\alpha] \leq \beta$ and this can only be if $B[A/\alpha] = \beta$. This is possible only if $B = \alpha$ and $A = \beta$. But the lemma above shows that I can contain only inequalities $A \leq \alpha$ in which A is not a variable. Thus, we have arrived at a contradiction, which shows that I can be ordered as claimed.

Now it is easy to write down directly a solution to the ordered inequations, simply by solving them from left to right. We define a solution τ iteratively as follows. Initially, τ is the empty function. Then, let i be the smallest index such that $\tau(\alpha_i)$ is not yet defined. Define $\tau(\alpha_i)$ to be $(B_1 + \cdots + B_l)\tau$, where $\{B_1, \ldots, B_l\}$ is the set $\{B \mid (B \leq \alpha_i) \in I\}$ (notice that α_i may be equal to some other α_j). The choice of order of the B_j and the associativity of the sum $(B_1 + \cdots + B_l)$ is not important. By the way we have ordered the inequations, it follows that the substitution $(B_1 + \cdots + B_l)\tau$ is well-defined. Moreover, at any time the partial definition of τ validates all the constraints $A_j \leq \alpha_j$ for which $\tau(\alpha_j)$ is already defined. Thus, by iterating this definition, we obtain a solution τ for I. Notice that we have thus show that a constraint set I of the above form is always solvable.

Steps 3 and 4 of the algorithm can be understood as an implementation of this method of solving I.

Thus we have shown that the algorithm finds a solution C if and only if such a solution exists. □