General homomorphic overloading

Alex Shafarenko
Dept. Comp. Science, University of Hertfordshire, AL10 9AB, U.K.
a.shafarenko@herts.ac.uk

No Institute Given

Abstract. A general homomorphic overloading in a first-order type system is discussed. Type inference is applied within predefined classes each containing an arbitrary first-order subtyping hierarchy. We propose a computationally efficient type inference algorithm by converting the attendant constraint-satisfaction problem into the algebraic path problem for a constraint graph weighted with elements of a specially constructed non-commutative star semiring. The elements of the semiring are monotonic functions from integers to integers (including $\pm\infty$) with pointwise maximum and function composition as semiring operations. The computational efficiency of our method is due to Klene's algebraic path method's cubic complexity. Our algorithm is applicable to type inference in the presence of unknown external types and supports distributed type inference.

1 Introduction

The concept of homomorphic overloading (h-overloading for short) is not completely new, although to the best of our knowledge it has not been laid into the foundations of any type system before. The original idea probably goes at least as far back as Reynolds's paper [6], where he remarked that "the key to ensuring that implicit conversions [1] and generic operators mesh nicely is to require a commutative relationship between implicit conversions and homomorphisms". To illustrate this, consider the following example. Let a generic operator f be defined on two types: $f_1 : a_1 \to b_1$ and $f_2 : a_2 \to b_2$, and let also $a_1 \sqsubset a_2$ and $b_1 \sqsubset b_2$. Under such conditions, the operator application $f\,x$ is naturally ambiguous. Indeed if $x : a_1$ it has the type a_2 as well so then which of the results $f_1\,x$ or $f_2\,x$ is expected? The usual principle is to choose the least type, i.e. that of f_1, so the result is $(f_1\,x) : b_1$. However this is coercible to b_2 which gives rise to the question: what is the relationship between the *value* of $f_1\,x$ raised to the type b_2 and the value of $f_2\,x$?

Reynolds suggests that the results for so overloaded operators must be the same. For instance, if we consider, following [6], $+_1 : (int, int) \to int$ and $+_2 : (real, real) \to real$ we find that $x +_1 y$ coerces to type real to give precisely the value of $x +_2 y$ (assuming that the available range of integers can be represented as

[1] i.e., coercions — A.S.

floats without rounding, which is usually the correct assumption). It is easy to see that in this example the coercion from integer to real serves as a homomorphism from $(int, +)$ to $(real, +)$, hence our term "homomorphic overloading". Paper [7] does not treat this homomorphism as a vehicle of type inference, but rather as a category-theoretical basis for formal semantics of a language that includes generic operators and coercions. By contrast, our concern is exactly the former.

In [4] we showed that a primitive form of h-overloading, where the type signature was constrained to fixed supertypes and subtypes of participating type variables, allowed fast type inference in the presence of unknown external types. The resulting types were inferred as explicit functions of the external types using the longest path algorithm on a constraint graph. We further showed the utility of h-overloading by giving an example of a language for stream processing that benefited from it. However, our solution was not generic, as it limited the variety of overloaded operators to a very restricted set of "offset-homomorphic" operators with a special type signature. Thus arbitrary h-overloading was not supported, in particular, there was no provision for arbitrary user-defined generic operators.

In this paper we shall lift restrictions on the h-overloaded signatures, which will make user-defined families of h-overloaded operators possible, while retaining the original complete inferability of types shown in [4]. We will introduce a combined overloading scheme which uses h-overloaded types within archetypes, which are groups of types belonging to disjoint subtyping hierarchies. This makes it possible to combine general type checking with automatic inference of homomorphic types.

The rest of the paper is organised as follows. In the next section we will review some of the basic concepts of the homomorphic type theory. Section 3 will introduce a new abstraction for defining type constraints: a star semiring of integer functions. We shall re-formulate the type inference problem as an *algebraic path* one and will find the solution to the former in terms of the latter. Section 4 focuses on the solution algorithms and implementation issues. Section 5 discusses applications of our method to various programming languages, and finally there are some conclusions.

2 H-overloading

Before introducing homomorphic overloading formally, we must note that h-overloading does not need to be the only overloading mechanism in a language that benefits from it. Indeed, one important reason to use overloaded operators is to avoid the proliferation of notation by reusing symbols based on their informal, mnemonic aspect. Where h-overloading is possible, it can be left implicit since, as we shall show, its disambiguation is always automatic and computationally efficient. By contrast, non-homomorphic overloading requires explicit declarations of type (or class of types, as in Haskell) since genuine ambiguity may arise when the program context does not constrain the choice of an overloading tightly enough. Using another of Reynolds's examples, if '+' were to denote both string concatenation and arithmetic addition, an assignment such as $a := b + c$, where all three variables are external to the program unit, would leave the type ambiguous, requiring an explicit declaration of type. H-overloading of the numerical instances of '+' would enable generic declarations like $a, b, c : numeric$ rather than requiring, for example, a more specific (and restrictive) $a, b, c : int$ but it would not eliminate type declarations completely, since the possibility for a, b and c to be of string type cannot be ruled out automatically.

Thus we consider types as being qualified by an 'archetype' specification, which is explicitly declared and is not subject to inference (although it is, of course, subject to type checking in a standard way). Here by archetype we mean a set of all subtypes of a well-defined type. For instance, numbers form an archetype with the usual subtyping into integers, reals and complex numbers; pairs of numbers form an archetype which contains a lattice of subtypes, etc.

One archetype may qualify several type attributes at the same time. For instance, numerical arrays can be assumed to have the following attribute structure:

$$narray(etype, rank),$$

where *narray* is an archetype of numerical arrays, which is declared, *etype* is the type of the array element taken from the subtyping hierarchy $int \sqsubset real \sqsubset complex$ and *rank* is the number of array dimensions taken from the hierarchy $0 < 1 < \ldots < r_{max}$, where the coercion from lower to higher rank is achieved by infinite replication of the corresponding array in the extra dimensions. This archetype was assumed in [4] in defining a stream processing language, where all operators were overloaded homomorphically in *etype* and *rank*. Another example could be the string archetype: $text(len)$, where *len* is the maximum size of the string, with obvious subtyping. Our subtyping scheme is, at the moment, first-order as we do not allow functional subtyping, the reason being that contravariance of function-argument types destroys the semiring construction described in Section 3, making type inference inefficient. This circumstance prevents our typing scheme from being used in a general functional language. We do nevertheless take full account of contravariance of non-functional types, making our approach applicable to first-order, single assignment languages, such as SAC[17] and ASTL[15], as well as imperative languages with atomic subtyping, notably Fortran. Here contravariance manifests itself in the *downward* coercion of an assignment target and is the reason that the least type of a variable is required to be sufficiently high.

In the rest of the paper, we shall assume the archetype qualifiers of all (sub)expressions in a program to have been deduced from the archetype declarations and the program text, so that they can be omitted from type signatures without creating an ambiguity. We also assume that two types can be in a subtype relation only if they come from the same archetype; in this sense all archetypes are disjoint. An n-ary operator is assumed to act on the Cartesian product of types, on which subtyping is defined in the standard way, i.e. component-wise.

Our focus will be on the inference of the least permissible types in a program where all operator overloadings are required to satisfy the following

Homomorphism restriction *For any (overloaded) operator F, an instance $F_2 : a_2 \to b_2$ is said to be homomorphic to an instance $F_1 : a_1 \to b_1$ iff $a_1 \sqsubseteq a_2$, $b_1 \sqsubseteq b_2$ and $(\forall x : a_2) b_{21} F_1 x = F_2 a_{21} x$, where a_{21} is the type coercion $a_1 \to a_2$ and b_{21} is the coercion $b_1 \to b_2$. For any overloaded operator F and any pair of its instances $F_{1,2}$ having identically qualified signatures, one instance must be homomorphic to the other.*

Proposition 2.1 *The set of identically qualified instances of an overloaded operator that satisfies the homomorphism restriction is linearly ordered.*

This follows from the fact that homomorphism is an antisymmetric relation, which is also transitive since the coercions are compositional, i.e. $(\forall t_1 t_2 t_3 : t_1 \sqsubseteq t_2 \sqsubseteq t_3) c_{31} = c_{32} \circ c_{21}$, where c_{ij} is the coercion $t_j \to t_i$). Note that the linear order of instances induces a linear order on the operand and result subtypes. This does not mean that the subtyping structure of an archetype must be a chain; it only has to *contain* a chain for every overloaded operator family defined on it. Thus, different operator families can potentially use different chains within the archetype without violating the homomorphism restriction. For any h-overloaded n-ary operator family F with k instances, we will write its type signature as follows: $F : \omega_1 \times \omega_2 \times \ldots \omega_n \to \omega_0$, where all ω_i are chains of length k in their respective archetypes. The potential confusion with the type signature of a single operator where ω_i are sets of *values* will be avoided by using small Greek letters only for chains of types. A type signature in this form does not by itself define the relationship between the output type of the operator family and its input types, it only defines the ranges of those types within their corresponding archetypes.

The homomorphic restriction has two important consequences. Firstly, it completely disambiguates operator application: $F\,x$ can always be interpreted as the application of the *lowest* instance of F compatible with the type of x. If the programmer meant a higher instance and applied a further operator to the result assuming that type, this is not a problem, since the result of applying the lower instance is coercible to the output type of the higher one, *yielding exactly the same value.*

Secondly, since Proposition 2.1 places the input and output types on chains in subtyping orders, in any well-typed expression the output type chain ω_1 of an operator F_1 belonging to the expression must *mesh* with the input chain ω_2 of the next operator F_2 up the expression tree. This means that, firstly, the output archetype of F_1 should be the same as the input archetype of F_2, which is not our concern since the archetype checking is assumed to have been done. Secondly, at least one element of ω_1 must be a subtype of some element of ω_2 so that the result of F_1 can be coerced to an input type of F_2. Let $x_{\max}$ be the greatest element of ω_1 coercible to ω_2:

$$x_{\max} = \max_{\omega_1} \{ x \mid (\exists y \in \omega_2) x \sqsubseteq y \} \,.$$

Then the operator F_1 can be restricted (without loss of generality) to just those overloadings for which the output type is at most $x_{\max}$. On the other hand F_2 can be restricted, also without loss of generality, to just those overloadings for which the input type is at most $x_{\max}$ (for arity 1). Similar conditions must be

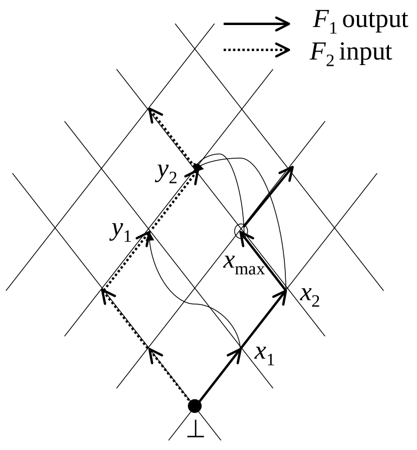

Fig. 1. Meshing type chains

satisfied in all operands of F_2 if its arity is greater than 1. Finally, a coercion map $c : \omega_1' \to \omega_2$ can be constructed:

$$c\,x = \min_{\omega_2}\{y \mid x \sqsubseteq y\}\,,$$

where

$$\omega_1' = \{x \mid x \in \omega_1 \wedge x \sqsubseteq x_{\max}\}\,,$$

and inserted between F_1 and F_2. It is obvious from the existence of $x_{\max}$ that for any $x \in \omega_1'$ the set on the right-hand side is nonempty, and so the function is well-defined. It is also easy to see that $c\,x$ is a non-decreasing function. Figure 1 gives an example of two meshed chains, where their common archetype is a lattice. The coercion map is depicted by curvy arrows: $c\perp = \perp$, $c\,x_1 = y_1$ and $c\,x_2 = c\,x_{\max} = y_2$

Another source of coercion is occurrences of program variables. When a variable occurs in a contravariant context, e.g. on the left-hand side of an assignment, the context defines a type chain (corresponding to the top-level operator on the right-hand side) and the type of the variable must be upwards of an output type belonging to that chain. The latter will be subject to type inference and is a priori unknown. Since there can potentially be several contravariant contexts in the program involving the same variable, the variable type must be the least upper bound of the corresponding output types. The variable may also occur in a covariant context, at which point the type derived from the contravariant contexts will be coerced up to the least member of the input type chain assumed by that covariant context.

The difference between meshing a variable with an operator and meshing two operators is subtle. The procedure exemplified in fig 1 effectively maps a chain onto another chain preserving the order, whereas in the case of variable-to-operator meshing, the least upper bound of the elements of the output chains is represented as a partially ordered subset of the archetype. A coercion map has to map this partially ordered subset onto the input chain of its associated operator. There is a useful factorisation, however, which reduces this kind of meshing to the previous kind. Let us consider the following example program

```
x := F (x,y);
...
x := G y
```

where $F : \alpha_1 \times \alpha_2 \to \beta$, $G : \alpha_3 \to \gamma$, the type of x, t_x is given by $t_x \sqsupseteq (b \sqcup g)$, where $b \in \beta$, $g \in \gamma$ are the (unknown) output subtypes of the operators. Note that depending on the shape of the β and γ chains within their common archetype, the least upper bound of b and g can sweep an arbitrary bounded subset, which does not have to be a chain.

For illustration, let us insert coercion functions into the program explicitly:

```
x := CxF F (CFx x, CFy y)
...
x := CxG G (CGy y)
```

Obviously, the output type of `CFx` is

$$\min_{\alpha_1}\{w \mid w \sqsupseteq (b \sqcup g)\} = \max(\min_{\alpha_1}\{w \mid w \sqsupseteq b\}, \min_{\alpha_1}\{w \mid w \sqsupseteq g\}),$$

which can be simplified to $\max_{\alpha_1}(c_b b, c_g g)$ where c_b and c_g are coercion maps of the kind discussed earlier. Observe that the agreement in type only involves operator output types, b and g with the type of the variable x being directly dependent upon them. Thus the types of program variables can be eliminated from the typing scheme; the output type variables of the corresponding top-level operators hold sufficient information.

In the general case the dependency of an input type of an operator on the output types of other operators via a variable has the form $\max_{i=1}^{n}(f_i\, x_i)$ for some n, where f_i is a map from a specific output chain to the common input chain. This construction is very important as it makes it possible to replace f_i by functions mapping a chain *offset* (which is a nonnegative integer representing the distance of a particular type along the chain from its bottom end) onto a chain offset. One can then reason about types solely in terms of those offset numbers. This follows from the factorisation exemplified above, i.e. from the fact that for any chain ω in a partial order P and any bounded set $S \subseteq P$

$$\min_{\omega}\{x \mid x \sqsupset (\bigsqcup S)\} = \max_{\omega}\{By \mid y \in S\}$$

where $B : P \to \omega$ is given by

$$Bx = \min_{\omega}\{y \mid y \sqsupset x\}$$

provided that such B exists.

Crucially, under h-overloading, a similar type representation exists for the operators themselves with respect to their multiple operands. It is given by the following

Proposition 2.2. *For any homomorphically-overloaded n-ary operator $F : (a_1, a_2, \ldots, a_n) \rightarrow b$, the output type offset $\hat{b}$ can be expressed as a function of the input type offsets $\hat{a}_i$ as follows*:

$$\hat{b}(\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_n) = \max(f_1\hat{a}_1, f_2\hat{a}_2, \ldots, f_n\hat{a}_n) \ ,$$

where $f_i : \mathbb{I}_i \rightarrow \mathbb{I}_0$ are some non-decreasing functions, $1 \leq i \leq n$, $\mathbb{I}_i = [0, k_i]$ is the offset range of the ith operand, k_i is the type offset of the highest overloading in the ith operand relative to the lowest overloading operand type, and $\mathbb{I}_0 = [0, k_0]$ is the output type offset range, with k_0 the difference between the maximum and the minimum output types along the subtype chain.

The proof of Proposition 2.2 follows from the observation that each operand separately demands a certain lowest overloading, and that it is also compatible with all overloadings higher than that one. Consequently, the least output type corresponds to the highest demand, which explains the maximum in the formula. The non-decreasing nature of the functions f_i comes from the fact that raising the type of ith operand along its chain can only make it too high for the current overloading and hence demand a higher one, with a higher output type.

For convenience, we extend the function domains so that $\mathbb{I}_i = \mathbb{I}_0 = \mathbb{Z} \cup \{-\infty, +\infty\} = \mathbb{Z}^\infty$ for all i and assume that $(\forall x < 0, i) f_i x = -\infty$ and $(\forall x > k_i, i) f_i x = +\infty$. The latter assumption models a type error by yielding an infinitely high supertype when the input type range is exceeded, and the former one is motivated by the semiring construction in Section 3. We shall call functions such as f_i and the above-mentioned coercion map c *type maps* when they are expressed in offset form $\mathbb{Z}^\infty \rightarrow \mathbb{Z}^\infty$. The range of $x \geq 0$ in which $f\,x < \infty$ is the *carrier* of the type map f. Since our type maps are based on finite subtype chains, we shall assume that all carriers are finite. The set of all such functions will be denoted as $\mathbb{F}$ below.

To summarise, the type analysis of a program written in a language with h-homomorphic operators breaks down into the following stages:

1. analysis of the explicit archetype declarations contained in the program.
2. analysis of the operator definitions, including the structure of h-homomorphism within each archetype.
3. archetype checking throughout the program
4. determination of coercion maps induced by meshing, with a subsequent conversion into offset form; elimination of variables by connecting co- and contravariant occurrences by type maps.
5. recording of all type signatures and converting them into offset form; recording of the type maps.
6. type inference

It is the last stage that we focus on in the next section.

3 Type Inference

A primary constraint set As usual, type inference begins with associating fresh type variables with all subexpressions in the program. In our case, these variables represent type offsets from $\mathbb{Z}^\infty$ rather than type values for the reasons explained earlier. For every operator occurrence, the operator type maps are invoked to produce a type constraint in the form:

$$v_0 = \max_{i=1}^{n}(f_i v_i)\,,$$

where $v_i,\ i = 0 \ldots n$ are any of the type variables just introduced. The constraints can be broken down into a set of simpler constraints in what we shall call *canonical form*:

$$\tau_0 \geq f_i\, \tau_i\,,$$

on the assumption that the minimum type assignment is sought. All canonical constraints in a program constitute the *primary constraint set.* This set can be assumed to contain exactly one constraint for every pair of types a and b. Indeed, if there are two constraints between these types, $a \geq f_1\, b$ and $a \geq f_2\, b$, then they can be replaced by an equivalent constraint $a \geq f_{1\oplus 2}\, b$, where for all $x \in \mathbb{Z}^\infty$, $f_{1\oplus 2}\, x = \max(f_1\, x, f_2\, x) = (f_1 \oplus f_2)\, x$. (We denote the operator of the pointwise maximum of two functions by $\oplus$.) On the other hand, if there are no constraints between a and b, then the constraint $a \geq \mathbf{0}b$ can be added, where $\mathbf{0} : \mathbb{Z}^\infty \to \mathbb{Z}^\infty$ such that for all $x \in \mathbb{Z}^\infty$, $\mathbf{0}\, x = -\infty$. Thus one can speak of an $n \times n$ constraint matrix C_{ij} defining the primary constraint set for n type variables. Each element of C_{ij} is the function $\mathbb{Z}^\infty \to \mathbb{Z}^\infty$ that occurs in the constraint between types x_i and x_j in canonical form.

Note that some of the type variable are associated with program variables which are external to the program unit being compiled and which are, consequently, not subject to inference. The purpose of type inference is to express the least type of each program variable as a function of those external types.

Constraint set expansion The simplest type inference procedure would be to initially assign 0 to all type variables associated with internal variables, and then iterate the constraint set until a fixed point is reached or a type variable acquires the value of infinity. In matrix form, we seek a solution to the constraint satisfaction problem $\boldsymbol{x} = C\,\boldsymbol{x}$ as a fixed point of the iterative process:

$$\boldsymbol{x}^{[0]} = \mathbf{0};\ \boldsymbol{x}^{[k+1]} = C\,\boldsymbol{x}^{[k]}\,.$$

Here $C\,x$ denotes $\bigoplus_{i=1}^{n} C_{ij}\, x_j$.

The procedure is sound, since at each iteration it delivers a lower bound of all types implied by the primary constraint set. Also, due to the non-decreasing nature of all matrix elements of C, at each iteration which does not deliver a fixed point, it produces an increased lower bound for at least some type variables. Since the carriers of all matrix elements are finite, a fixed point exists and is reachable.

Obviously, the constraint set is satisfiable iff none of the lower bounds delivered at the fixed point is infinite.

This solution has two potential problems. First of all, the number of iterations is only bounded from above by the total length of all type chains, since at each iteration (which does not result in a fixed point) only one type variable has to increase. Secondly, since the numerical values of the external type parameters are unknown, iterations have to be performed with the matrix C by raising it to a power (using function composition as multiplication and $\oplus$ as addition). This by itself is a costly operation, to perform even once.

We propose a more efficient algorithm, based on the algebraic path problem, which we consider next.

Algebraic structure of $\mathbb{F}$ Recall that the elements of the constraint matrix are drawn from the set $\mathbb{F}$ of nondecreasing functions $\mathbb{Z}^\infty \to \mathbb{Z}^\infty$ that yield $-\infty$ on all $x < 0$ and $+\infty$ on sufficiently large $x \geq 0$. Consider a six-tuple $\Phi = (\mathbb{F}, \oplus, \odot, *, \mathbf{0}, \mathbf{1})$ where $\oplus$ is as defined above, $\odot : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ is a function composition, $* : \mathbb{F} \to \mathbb{F}$ is Kleene's star operation:

$$f^* = \mathbf{1} \oplus f \oplus (f \odot f) \oplus (f \odot f \odot f) \oplus \ldots,$$

$\mathbf{0} \in \mathbb{F}$ is as defined above and $\mathbf{1} \in \mathbb{F}$ is the identity function[2]: $\mathbf{1}x = x$ for $x \geq 0$, $\mathbf{1}x = -\infty$ otherwise.

Proposition 3.1 *$\oplus$, $\odot$ and $*$ are closed in $\mathbb{F}$.*

Indeed, the $\oplus$ operation is closed in $\mathbb{F}$ since the point-wise maximum of two nondecreasing functions is a nondecreasing function, whose carrier is included in the union of the carriers of the arguments and so is finite. Likewise, the composition of two nondecreasing functions is a nondecreasing function. The behaviour of this function at negative arguments and $\pm\infty$ is proven immediately by substitution; the carrier of the result is the same as that of the first operand, so $\odot$ is closed in $\mathbb{F}$.

Finally, the star operator is defined in terms of the fixed point of a series, each member of which is computed from elements of $\mathbb{F}$ using the operators $\oplus$ and $\odot$. Since they are both closed in $\mathbb{F}$, the star operator itself is closed in $\mathbb{F}$ if the fixed point exists. The fixed point does exist, since the series of partial sums is point-wise nondecreasing and since $\mathbb{Z}^\infty$ includes $+\infty$. In fact, we will show below that the fixed point can be computed in a finite number of steps by an efficient algorithm, which means that the series for the star operator is always finite for any element of $\mathbb{F}$. This obviates the proof that the star construct is well behaved; such a proof would usually be required for an infinite star series. ‡

Proposition 3.2 $(\mathbb{F}, \oplus, \mathbf{0})$*and*$(\mathbb{F}, \odot, \mathbf{1})$ *are monoids, the former is commutative.* Indeed, function composition is associative and so is point-wise maximum. The elements $\mathbf{0}$ and $\mathbf{1}$ are obviously the identities of the respective operations. ‡

[2] Strictly speaking the identity function is not in $\mathbb{F}$ since it does not have a finite carrier; nor is $\mathbf{0}$. However, we include them in $\mathbb{F}$ as special elements. The use of both $\mathbf{1}$ abd $\mathbf{0}$ with $\oplus$, $\odot$ and $*$ does not lead to further infinite-carrier elements.

Proposition 3.3 *Operation $\odot$ distributes over $\oplus$ both on the left and on the right:*

$$a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c) \text{ and } (b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)\,.$$

The proof is by point-wise application, using the nondecreasing nature of functions a, b and c. ‡

Proposition 3.4 $\mathbf{0}$ *is a null with respect to $\odot$: $\mathbf{0} \odot x = x \odot \mathbf{0} = \mathbf{0}$* The proof follows immediately from the construction of the element $\mathbf{0}$. ‡

Propositions 3.1-4 form the proof of the following

Lemma 3.5 *Φ is a star semiring.*

Inference procedure Now consider the constraint satisfaction problem again. Let us associate every type variable with a vertex of a weighted, directed graph G. Each edge (v_i, v_j, f) of the graph represents the constraint

$$v_i \geq f\, v_j\,.$$

Since the (internal) program variables occur in both covariant and contravariant contexts, the graph is not necessarily acyclic, and may contain infinite as well as finite walks. Each walk corresponds to a chain of primary constraints connecting its ends, and hence to a *secondary* constraint corresponding to the (finite or infinite) $\odot$-product of the weights of the participating edges. The tightest constraint between any types v_i and v_j due to the primary constraint set is the $\oplus$-sum of the weights of all walks W_{ij} in graph G from vertex i to vertex j:

$$P_{ij} = \bigoplus_{w \in W_{ij}} \Big(\bigodot_{i \in w} f_i\Big)\,,$$

where the selection of vertices from the walk w in the $\odot$-product is in the walk order. This is a formulation of the classical *algebraic path problem* [11] for the semiring Φ and graph G.

The solution to the algebraic path problem is the matrix P_{ij} of semiring values. We will define an efficient algorithm for its computation below. For now let us assume P_{ij} has been computed, and proceed to the type assignment.

Proposition 3.6. *Divide the set of type variables $\{v_k \mid 1 \leq k \leq n\}$, into external ones $k \leq n_e$, which are not subject to type assignment, and the rest $n_e < k \leq n$. The least type assignment is given by the following formula:*

$$v_k = \min_{v_k^*}\{x \mid x \geq \max_{i=1}^{n_e}(P_{ki}\, v_i)\} = P_{kk} \odot \max_{i=1}^{n_e}(P_{ki}\, v_i)\,,$$

where v_k^* is the set of solutions of the equation $x = P_{kk}\, x$. The outline of the proof is as follows. First of all, observe that any type assignment for the variable v_k has to satisfy the secondary type constraint $v_k \geq P_{kk} v_k$. Since $P_{kk} \geq \mathbf{1}$ point-wise (since at any rate $v_k \geq v_k$), only the fixed points of P_{kk} are suitable as potential type assignments for v_k. Secondly, v_k must be large enough to satisfy all primary and secondary constraints induced by the external types, which explains the above formula. The third part of the equation is due to the fact that P_{kk} is the point-wise maximum of all cyclic chains on vertex k, hence $P_{kk} \odot P_{kk} = P_{kk}$ and so, for all $x \in \mathbb{Z}^\infty$, $P_{kk}(P_{kk}\, x) = P_{kk}x$. This means that $P_{kk}x$ is a fixed point of P_{kk}. The fact that this fixed point is the least one greater than or equal to x is due to $P_{kk} \geq \mathbf{1}$ point-wise and to its nondecreasing nature.

One might think that v_k must be large enough to satisfy the constraint induced by any other internal variable v_j: $v_k \geq (P_{kj}v_j)$. We claim that this happens automatically. Indeed, assume the contrary, i.e. that for some j, $v_k < (P_{kj}v_j)$. By the above assignment $v_j \geq P_{jj} \odot P_{ji}v_i$ (recall that P_{jj} is a nondecreasing function, so it distributes over the maximum), and so $v_k < P_{kj} \odot P_{jj} \odot P_{ji}v_i$ for any external v_i. The right-hand side reduces to $P_{ki}v_i$ by definition of P and semiring distributivity. Hence $v_k < P_{ki}v_i$, which contradicts our type assignment and proves its validity. ‡

4 Implementation

The type inference method proposed in the previous section requires the ability to compute the algebraic path matrix P_{ij} efficiently. This is achieved by Kleene's algorithm in $O(n^3)$ semiring operations using the following iterative process. Set the initial value $P_{ij}^{[0]}$ according to the primary constraint graph. For any edges (i, j) not found in the graph set $P_{ij}^{[0]} = \mathbf{0}$. For $k = 1 \ldots n$ do:

$$(\forall i, j) P_{ij}^{[k]} = P_{ij}^{[k-1]} \oplus (P_{ik}^{[k-1]} \odot (P_{kk}^{[k-1]})^* \odot P_{kj}^{[k-1]})$$

The solution is $P_{ij} = P_{ij}^{[n]}$.

At each iteration, the algorithm requires $2N^2$ semiring multiplications and N^2 semiring additions as well as one star operation. We consider the implementation of those next.

We propose the representation of semiring elements as sorted lists of pairs (a, v) where $a \geq 0$ is the value of the function argument and v is its result. The list is sorted in the ascending order of a. The value of the function for the arguments greater than the last one listed are assumed to be $+\infty$. The empty list corresponds to the maximum element of Φ, $\phi_{\max}$: $(\forall x \in \Phi) x \oplus \phi_{\max} = \phi_{max}$. The elements $\mathbf{0}$ and $\mathbf{1}$ are represented as special values recognised by all three operators.

It is easy to see that the $\odot$ operation in this representation is little more than the classical database *join* of the operands equating the v field of the first operand and the a field of the second; it yields a sorted list as a result. Both source lists are only traversed once, thanks to the nondecreasing nature of the semiring elements and the fact that any emerging lists are already sorted. The $\oplus$ operator is implemented as a join in the field a of both lists followed by the pointwise maximum of the corresponding v fields. Of course the a field does not even need to be stored, as it contains merely the sequential number of the list element.

The star operator is slightly trickier to implement. Observe that since Φ is idempotent (i.e., $(\forall x \in \Phi) x \oplus x = x$), $(f \oplus \mathbf{1})^* = f^*$, which can be proven by substitution. Hence without any loss of generality we can assume that $f\,x \geq x$ for all nonnegative x. The first step is to identify closed intervals of x, $[b_i, e_i]$ such that:

$f\,(b_i - 1) \leq b_i - 1$,

$f\,k > k$ for $b_i \leq k < e_i$ and

$f\,e_i = e_i$.

If no such interval exists, it is easy to see that $f\,x = x$ for all $x \geq 0$, in which case $f^* = f = \mathbf{1}$. Indeed, since for any $f \in \Phi$, $f(-1) = -\infty$ and $f(+\infty) = +\infty$, there is at least one suitable pair of e_i and b_i. Hence the middle condition is not satisfied, which means that for all k $f\,k \leq k$, hence $f \oplus \mathbf{1} = \mathbf{1}$.

In the general case, the carrier of f is partitioned into one or more closed intervals of the above sort with possibly intervals where $f\,x \leq x$ occurring in between those. We then apply the following

Proposition 4.1. *Within each interval* $[b_i, e_i]$*,* $f^*\,x = e_i$*.*

Indeed acting f on any point within the interval will produce a greater result not exceeding e_i (which is the value of a nondecreasing function at the right end of the interval where it is nondecreasing, hence the maximum). Therefore, repeated application of f will eventually reach e_i which is a fixed point.

The star algorithm should consequently proceed in two passes. In the first pass, the closed intervals are identified by scanning the list and comparing the current and previous elements. At the same time any elements for which $v < a$ are adjusted to $v = a$. In the second pass, the answer is computed by filling up the intervals with their final value of a. This is best accomplished by placing the list elements on top of a stack during the first pass, and reading them off the top of the stack in the second, so that the ends of intervals could propagate backwards.

One last observation: in the previous section we stated that f^* maps any x to the nearest fixed point equal or exceeding x. Clearly our algorithm has this property.

From the description of the semiring algorithms, it is clear that their computational cost is $O(L)$ where L is the length of the longest chain in the subtyping system. An obvious optimisation would be to exploit the fact that there are usually much fewer instances to an operator than there are different subtypes in a type. Consequently, the type maps are likely to be step functions with many different a corresponding to the same v. The above algorithms can easily be modified for such functions: only the first record with the same v need be kept, the join algorithm must compare for $\geq$ instead of equality, etc. As a result the computational cost of semiring operations could be reduced to $O(V)$ where V is the maximum number of overloadings defined for any operator in the program.

5 Applications

The type scheme introduced in this paper was developed as part of the ASTL project [15], which is based on a Grid-aware stream-processing language ASTL. ASTL describes nodes of a stream-processing network which are connected by channels, called "streams". Each channel is a statically typed entity, capable of carrying a certain structure of records. Several streams can be fed into a node, which can produce output streams as well. The nodal program of ASTL is called a "stream transformer" and is presented as a combination of an *interface* and a set of *stream recurrence relations* which describe how data is processed. The interface names input and output streams and defines necessary information for matching the record structures against user-defined patterns in order to extract elementary fields. The fields are treated as typed entities, which are subject to subtyping.

Since ASTL was proposed for distributed numerical computing on the Grid, it supports a hierarchy of numerical types, from Boolean to complex, which it treats as a single type chain. Binary arithmetic operators, such as (+) act on a chain in a tuple type lattice, having homomorphic overloadings as follows:

```
+:(cmplx,cmplx)->cmplx
+:(real,real)->real
+:(int,int) -> int
```

Here each overloading is homomorphic to the one above it with respect to the argument and result coercions. Moreover, since ASTL is intended for computational applications, it treats each variable as potentially array-valued. The array rank, i.e. the number of dimensions, is treated as a static type attribute, again with subtyping. Arrays are subtyped by infinite replication in further dimensions. For instance, a replicated scalar is a subtype of a general 1-dimensional array, a replicated 1d array is a subtype of a 2d array, etc. This approach allows natural specialisation of modules with respect to translational symmetry of array data: for instance, a matrix dot product can be treated as two rank coercions (from 2d to 3d), one for either operand, a generic 3d multiplication followed by a +-reduction in the third dimension. This way unnecessary proliferation of multi-dimensional versions of array operators is avoided while providing a complete array toolkit, not dissimilar in expressive power to the APL. Array subtyping is treated strictly homomorphically, just like the numerical subtyping. This means that an operator applied to low-dimensional arrays would act consistently on those operands' higher-dimensional versions as well. Not only does this save notation, it also makes the program far more readable compared to the cryptic style of APL, without reducing the possible variety and functionality of specific array-processing operations.

The body of the stream transformer defines the relations between the input and output stream values as a set of pure, tail-recursive functions, which are defined in the form of a single assignment to a stream variable. It uses the numerical and array forms of subtyping for specifying a transformer as a set of generic recurrence relations, which can be specialised by the compiler automatically whenever the environment types become known. Those types are ones associated with the input streams in the interface section of the module. They are set by the Grid environment when transformers are connected up by streams. However, thanks to the theory presented in this paper, the transformers can be fully analysed individually, *before* this happens. Indeed, it is a requirement in ASTL that each overloaded operator that occurs in a recurrence relations inside a transformer satisfies the homomorphism restriction. There is no support for higher-order functions, so arrow types do not occur in user-defined constructs, and thus arrow-type contravariance is avoided. There are, however, assignments to variables (if only occurring once), and so the contravariance of an assignment target is present. The user can specify arbitrary homomorphically-overloaded functions as families of external subroutines linked to a stream transformer.

To summarise, all the enabling conditions of the proposed general homomorphic overloading scheme are satisfied in ASTL. Therefore, all output types exported by the interface of a transformer are bound by Proposition 3.6 to some nondecreasing functions of the input types. The functions in question can be derived from the source of an individual transformer by applying the type inference procedure described above. At this stage, despite the lack of external type information, errors can be diagnosed and the maximum allowable type of the input streams can be ascertained from the diagonal elements of the constraint matrix. A type error occurs when the least allowable value of a type variable is $+\infty$, or when the maximum allowable type of an input stream type is $-\infty$. The graph-theoretical nature of Kleene's algorithm, which is at the core of our type-inference procedure, helps to trace back the provenance of errors in order to produce readable diagnostics for the programmer [16].

Another unique feature of the proposed homomorphic type scheme is the availability of distributed type inference, which we discuss next.

Distributed type inference In our earlier work [4] we stated that flexible type systems are especially useful in distributed environments where program modules often represent generic services offered to other modules communicating over a network. In a distributed computing environment, modules are often deployed on different hosts and are connected by data streams, which can be represented as variables common to two or more modules. Input stream variables occur in expressions inside the module but not on the left-hand side of an assignment or in other contravariant contexts. Their type is imported from the module where these variables are defined and made available for export.

Due to the direction of type coercions, the types of input variables are only bounded from above, while the types of the output variables are generally also bounded from below as they do occur in contravariant contexts. The stream leaving one module and coming to another can be coerced up at the receiving end or rejected. The whole system of interconnected modules must come to a common type assignment before starting to operate, and after any reconfiguration that replaces, removes or adds modules.

The procedure outlined in the previous section obtains a relation between input and output types of every module whereby the type of the variables in the output interface is dependent on the type of the variables in the input interface, the latter considered a given. The next task is to find the type assignment for all stream variables where the joint constraint set of all modules is satisfied and no type of a stream variable can be lowered without violating one or more of the constraints.

Stream reconciliation As modules are connected with data streams, it is tempting to use these for conveying type information prior to run time. The following distributed type-reconciliation procedure is proposed.

Each module approximates its input type variables by zero and calculates the type of the output variables using Proposition 3.6. These output types are passed along the data links to the receiving module in a special message. When a module receives such a message, it treats the received type values as a new approximation of the input type variables. It then generates a new approximation of the output types to be sent to the corresponding inputs, etc.

Since the input types increase with each approximation, the output types stay the same or increase as well. If the output types of the module remain the same after an approximation, a termination point is reached: no more messages will be sent by this module. It is easy to see that after a finite number of messages have passed between the modules, each module either reaches a termination point or receives types that makes one of the output types infinite. In either case the inference process terminates.

The modules in a module network need to know that they all have reached a termination point and whether or not errors have been encountered. The former task is an instance of a well-known distributed termination-detection problem (see survey [8] for existing algorithms). The latter task can either be incorporated in the termination detection infrastructure or it can be completed separately by back-propagation of failure to the neighbouring nodes.

The problem with stream reconciliation is that its complexity depends on the size of the longest path in the module dependency graph as well as the range of types. A long path and a large type range could cause reconciliation to be very slow. Next we consider a faster method which has been inspired by a known message-routing strategy in data networks.

Bellman-Ford relaxation

We use as a basis the Bellman-Ford algorithm [9], which determines (shortest or longest) distances from all nodes of a routing network to a single distinguished node, and propose a similar strategy for distributed type inference.

According to the type inference procedure of the previous section, the type of an internal variable is the "greater" (in the sense of $\oplus$) of the type due to the module's own constraints and that due to the constraints imposed by the external variables. The following asynchronous parallel implementation follows the sketch in [13].

For each module m:

initially For each output type variable t_i broadcast the value of the least fixed point of P_{ii} to all modules s such that the variable t_i is exported from m to s.

relaxation step Upon receiving a type message on any of the input links, set the corresponding t_j and then using Proposition 3.4 recompute all t_i that are exported to other modules. If, as a result, any of the t_i variables have changed, send the new types to all the modules that these variables are shared with. If the received type message exceeds the maximum admissible type for the corresponding type variable and thus causes an infinite type to be produced, a reject message is generated and sent back to the originator. The latter is then in a position to forward this message further back, if necessary until it reaches the source of the contradiction.

This solution has two problems. One has already been encountered in the reconciliation procedure, namely that we require a termination detection mechanism to determine the point when no further corrections to the type variables can be made. The second problem is more serious. The asynchronous version of Bellman-Ford is known to exhibit exponential complexity in arbitrary graphs [13]. Fortunately in any component-based system there is usually some form of centralised control. The solution to our problem is to use the central control point to enforce synchrony on the type inference procedures operating at individual components. Namely, the relaxation step is synchronised with receiving a request from the central control point, and, after completing the step, each component sends a confirmation message back to the control, which triggers the next round after all confirmations have been received. Obviously, every component must now send its output type messages whether the output types have changed during the step or not.

The complexity of the synchronous Bellman-Ford algorithm is linear in the number of components so the termination point is guaranteed to be reached realistically fast.

Further applications Our overloading scheme has potentially a greater range of applications despite the lack of support for higher-order functional types. One of the interesting challenges would be to apply the idea of type homomorphism to an imperative language, e.g. Fortran, where subtyping of numerical types is semi-explicit: variables have to be declared as real, integer, etc., but operators, such as +, implicitly recognise subtyping hierarchies. It would be quite useful to be able to introduce a single type "number" with a subtyping hierarchy taking care not only of the various classes of numbers, but also various levels of precision, range etc. The programmer would not necessarily need to know any of the subtyping rules for an operator, being reassured by the type system that if the external data for a given subroutine are correctly typed then the rest of the variables will be specialised as appropriate by the compiler without loss of generality. That includes not only subexpressions, which, obviously, are typed by the compiler anyway, but, importantly, any variables defined *inside* a module. It is easy to see, that the type inference procedure we have proposed will work here as well, as long as the homomorphism of operator overloading is enforced (which can be done by slightly re-defining all operators and intrinsic functions) and provided that there is no higher-order functions used as operators — which is the case in most imperative languages. In fact, even the explicit declaration of external types can be avoided by type reconciliation similar to the one described above; it can be shown that in the absence of recursion (a common case in Fortran) external types can be reconciled by a single sweep across the procedure call graph. When recursive subroutines are used (and Fortran requires that any such subroutines are explicitly declared recursive), one could either demand an explicit type declaration, or resort to a fixed point calculation, which is outside the scope of this paper.

6 Discussion and Related Work

The issue of type inference with atomic subtyping has a long history. We cite papers [1, 2, 7] as ones where foundation work was done. However, the main thrust of this work is towards treating homomorphism of types. This issue was not approached systematically until a simplified theory was given by us in [4]. Our concept of type homomorphism is consonant to Lievant's idea of "discrete polymorphism" proposed in [3], where it was suggested that overloadings should be treated as models of a single theory. We believe that h-overloading is less restrictive as it allows higher instances to "expand" the functionality of the lower ones without destroying the consistency between them.

Technically, the most relevant to our work is the paper by Rehof and Mogensen [13], where a method is described for what they termed a "definite constraint satisfaction problem". Here all constraints are presented in a form similar to ours: $v_0 \geq f(v_1, \ldots, v_k)$, where f is a nondecreasing function. Then an algorithm is presented, with a complexity linear in the number of constraints, (i.e. quadratic in the number of variables n) which finds the least solution. The main difference is that in [13] the system of constraints is assumed to be *closed*, i.e. all variables are subject to type minimisation within the constraints. In our work we approach a more general problem of constraint satisfaction with external parameters so that the solution is a function of those. Immediately the domain ceases to be a semilattice and the algorithm from [13] becomes inapplicable. We have proposed a slightly more costly solution, with the cost $O(n^3)$, but which allows external types to be parameters in the type assignment. Here n is the number of type variables; since one can assume there is no more than one constraint involving a pair of type variables, the number of constraints does not exceed $n_c = n^2$, giving a complexity estimate of $O(n_c^{3/2})$. This measure is not dramatically worse than Rehof-Mogensen's, and certainly is not bad enough for the typical number of type variables in a module to render type inference unfeasible.

In reality, $n_c \ll n^2$ since it can be shown that most type variables are localised inside expression subtrees, each participating in precisely two constraints: one constraining the root of the subtree with respect to its children, and one where the root participates as a child of its parent tree. The only type variables that deviate from this pattern are those associated with actual program variables (as opposed to subexpressions), which are assigned, exported, etc. Since there are not too many of them in an individual module (perhaps a few hundred at worst), our version of Kleene's algorithm is likely to be computationally feasible.

In graph-theoretical terms, the relationship between [13] and our method is similar to the relationship between Dijkstra's shortest path algorithm and Folyd-Warshall's all-pair shortest path: the Dijkstra algorithm computes positive shortest distances from a single graph vertex (the bottom of the lattice order in Rehof-Mogensen's procedure), while the Floyd-Warshall technique computes minimum (or maximum) pairwise distances and does not require the edges to be positively weighted.

7 Conclusions

A type inference solution for a general first-order, atomic subtyping with homomorphic overloading has been proposed. We have shown that after archetype checking, an h-overloaded operator produces type constraints characterisable by nondecreasing functions on the expanded integer set. A star semiring Φ was proposed to capture algebraic properties of such functions. Using Φ, we have built a type inference procedure based on Kleene's algorithm. The procedure infers the least types of all internal variables in the program as explicit functions of the external types. We have discussed possible applications of the method, including those in an open environment, where type inference is distributed.

Future work will proceed in two directions. First of all, the proposed type inference technique will be applied to more languages. In the first place, we are planning the application to the language SAC (single-assignment C), see [17]. SAC supports a very general view on arrays, similar to APL and as such can benefit from a disciplined implicit subtyping. The second direction is towards easing the limitations of our method. We wish to support higher-order functional types for which contravariance currently prevents the semiring construction described in this paper. The consequence of this is that the "sum of products" for P_{ij} loses the benefits of distributivity and hence becomes combinatorially large. We believe a possible approach to this may be in terms of graph partitioning into purely covariant subgraphs connected by contravariant edges, where the fact that there would typically be a small number of such edges might lead to a computationally-feasible algorithm.

References

1. L.Cardelli and P.Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471-522, 1985.
2. J.Mitchell. Type inference with simple subtypes. *Journal of Functional Programming,* 1:245–285, 1991.
3. D. Lievant Discrete Polymorphism. *Proc. 1990 ACM Conference on LISP and Functional Programming*, pp. 288–297, 1990.
4. A. Shafarenko. Coercion as homomorphism: type inference in a system with subtyping and overloading. PPDP'2002, Pittsburg, PA. October 6-8, 2002

5. A.Shafarenko. RETRAN: a Recurrent Paradigm for Data-Parallel Computing. *Computer Systems Science and Engineering*, vol 11, No 4, July 1996, pp 201-209
6. J.C. Reynolds. Using category theory to design implicit conversions and generic operators. In: *SemanticsDirected Compiler Generation*, LNCS vol 94, pages 211-258. SpringerVerlag, 1980.
7. J.C.Reynolds. Three approaches to type structure. In: *TAPSOFT proceedings, LNCS* vol 185, pp.97-138, 1985.
8. J.Eifrig, S.Smith and V.Trifonov. Type inference for recursively constrained types and its application to OOP. *Theoretical Computer Science,* December 1995, vol. 152, no 2, p. 326–345.
9. J. Matocha and T. Camp, A Taxonomy of Distributed Termination Detection Algorithms, *The Journal of Systems and Software*, vol. 43, no. 3, pp 207-221, 1998.
10. L. R. Ford and D.R. Fullkerson. Flows in Networks. Princeton University Press, Princeton NJ, 1962
11. G. Rote, "Path Problems in Graphs", in G. Tinhofer, E. Noltemeier, M. Syslo (eds.), Computational Graph Theory, Springer-Verlag, Computing Suppl. 7, Wien, 1990, pp. 155–198.
12. Y. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science* 73:155-175, 1990.
13. J. Rehof and T. Mogensen. Tractable constraints in finite semilattices. In R. Cousot and D. Schmidt, editors, Proc. of 3rd Int. Static Analysis Symposium (SAS'96), pages 285–300. Springer LNCS vol. 1145, 1996.
14. N. Lynch. Distributed Algorithms. Morgan Kaufmann, 1996. p. 507-509.
15. A. Shafarenko. Stream Processing on the Grid: an Array Stream Transforming Language. *SNPD 2003*, pp. 268-276
16. J. Hansen and A. Shafarenko *Error reporting in a type system with homomorphic overloading.* University of Hertfordshire Department of Computer Science. Internal Report.
17. Sven-Bodo Scholz: Single Assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.* 13(6) pp. 1005-1059 (2003)