Coercion as Homomorphism: Type Inference in a System with Subtyping and Overloading

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

ABSTRACT

A type system with atomic subtyping and a special form of operator overloading, which we call offset-homomorphism is proposed. A set of operator overloadings is said to be offset-homomorphic when for each pair of overloadings the coercion function realises a homomorphism of types and at the same time certain conditions on the operator type signature are satisfied. We demonstrate that offset-homomorphic overloading has sufficient power for supporting a comprehensive set of array operations in a declarative language. The problem of inferring the least types in our type system is proven to be equivalent to the shortest path problem for weighted, directed graphs with non-negative cycle weights, which has a computationally efficient solution.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Polymorphism, Data types and structures

General Terms

Languages, Theory

Keywords

type inference, overloading, subtyping, data-parallel programming, array processing

1. INTRODUCTION

The interplay between overloading and subtyping has been an active research topic ever since the foundation work on subtyping was done by Cardelli[1], Mitchell[2] and Reynolds [6]. The intention of overloading in programming languages is to eliminate excessive notation, which is achieved by reusing operator symbols to denote different functions depending on the type of the operands. Despite the fact that denoting different functions by the same symbol creates confusion where there is no similarity between the operations, overloading schemes typically do not require different instances to be related in any way. For example, the use of the symbol + to denote numerical addition and string concatenation is typical in many languages, both declarative and imperative (Java and C++ are among the many examples). Apart from the fact that these operators are both associative, they are completely independent. On the other hand, the two instances of the addition operation, $+_1 : (int, int) \to int$ and $+_2 : (real, real) \to real$ are closely related algebraically.

The opposite concept of overloaded operator as a set of models satisfying a single theory was proposed by Lievant[3] and termed "discrete polymorphism". Here an operator is associated with a set of axioms which all its instances must satisfy. This makes all overloadings identical algebraically. In practice, discrete polymorphism may be too restrictive since the purpose of overloading is to be able to denote with the same symbol essentially different, albeit related, instances of an operation.

Returning to the example mentioned above, we note that the coercion $int \to real$ acts as a homomorphism of models $(int, +) \mapsto (real, +)$. In our opinion, this form of overloading deserves systematic treatment. Besides the numerical types, where homomorphism manifests itself in many operations and elementary functions due to the subtyping relation between numbers: $int \subseteq real \subseteq complex$, homomorphism plays an important role in multidimensional array types. Here a rich variety of operations can be defined on top of a subtyping hierarchy based on array rank (i.e. the number of dimensions): low-dimensional arrays are coerced up by replicating their elements in extra dimensions. Array functions, such as reductions and elementwise applications, naturally lend themselves to overloading as they apply to all array ranks. Without such overloading, array notation becomes unnecessarily detailed making the meaning of expressions hard to grasp. On the other hand, the use of operator overloading *and* operand subtyping makes the expression ambiguous — unless, as we shall demonstrate, operator homomorphism under type coercion is enforced.

The problem of overloading array operators becomes even more important when array-processing services are to be "discovered" on a Computational Grid [4]. Grid resources are advertised in on-line directories and can be allocated automatically to an ongoing computation by mobile agents. In order to determine the applicability of a resource, its declarative wrapper needs to be matched with the declarations of data objects or other processing resources. Matching by type and selecting a correct instance of an overloaded function would be an attractive strategy as it usually delivers some semantic guarantees. Such a strategy would require

PPDP'02, October 6-8, 2002, Pittsburgh, Pennsylvania, USA.

distributed type checking at the very least, but preferably a distributed type inference mechanism.

In this paper we focus on the problem of type inference in a declarative, data-parallel array language ASTL[1] with primitive recursion as the principal evaluation mechanism, a follow-up from an earlier project [5]. In the next section we will introduce our concept of overloading in the presence of subtyping, which exploits the homomorphism of overloaded operators with respect to type coercion. We then propose a simplified version of general homomorphism, which we call offset-homomorphism, and claim this version has sufficient power to support a significant class of applications while also supporting type inference. Section 3 justifies part of our claim by describing a comprehensive set of ASTL array operations which are typed using offset-homomorphism. Section 4 focuses on the remaining part of the claim, i.e. on the type inference method, showing that one exists which delivers exact least types or detects an inconsistency in polynomial time. Section 5 discusses the implications of our approach to distributed computing and finally we give some conclusions.

2. OFFSET HOMOMORPHISM

As mentioned in Introduction, the idea of homomorphic overloading in the presence of subtyping is central to our approach. Specifically, we concern ourselves with a language where overloaded operators must satisfy the following

Homomorphism restriction *For any (overloaded) operator L, an instance $L_2 : a_2 \to b_2$ is said to be homomorphic to an instance $L_1 : a_1 \to b_1$ iff $a_1 \subseteq a_2$, $b_1 \subseteq b_2$ and $(\forall x : a_2) b_{21} L_1 x = L_2 a_{21} x$, 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 L and any pair of its instances $L_{1,2}$, one instance must be homomorphic to the other.*

Lemma 1.1 The set of instances of an overloaded operator which satisfies the homomorphism restriction is linearly ordered.

Proof. Observe that if L_1 is homomorphic to $L_2 \neq L_1$, the converse is not true, since subtyping is an antisymmetric relation. Moreover, homomorphism of instances is a reflexive and transitive relation (assuming that the coercions are compositional, i.e. that $(\forall t_1 t_2 t_3 : t_1 \subseteq t_2 \subseteq t_3) c_{31} = c_{32} \circ c_{21}$, where c_{ij} is the coercion $t_j \to t_i$). Consequently, the instances of any operator form a poset. Since the homomorphism restriction requires every pair of instances to be homomorphic, the order is, in fact, linear. $\triangle$

We resolve the type ambiguity of an operator application $L\,x$ by defining it to be the result of applying to x the least instance of L, $L_{\min} : \tau \to \alpha$ such that the type of x is a subtype of τ. This means that each operator application generally consists of the coercion that raises the operand type to the input type of the nearest instance, and the application of that instance. There is, therefore, no least operand type, but there can be a maximum permissible operand type which corresponds to the highest instance of the operator. On the other hand, the output type can be increased further by coercion up to the maximum available type if the context requires it; it is therefore the *least* output type that we concern ourselves with.

Note that our choice of a disambiguation rule makes operator overloading completely transparent to the programmer: when applying an overloaded L to $x : a$, if a higher instance $L_2 : a_2 \to b_2$ is meant rather than $L_{\min} : a_1 \to b_1$, $a_2 > a_1 \geq a$ (and the result type $b_2 > a_2$ is expected by the rest of the expression) the programmer will not notice the difference. Indeed, thanks to the homomorphism of L_2 to $L_{\min}$, $L_{\min}\ x$ can be coerced to the type b_2 yielding exactly the same value as $L_2\ x'$, where x' is the value of x coerced to the type a_2.

The type hierarchy in applications is often consistent with computational cost, which means that choosing the least operator instance also minimises the complexity of computation. For example, it is generally much cheaper to compute an array result once and then replicate it than it is to perform a number of identical computations when a replicated result is required.

Corollary (type monotonicity): *The result type of a homomorphically overloaded operator is a non-decreasing function of a single operand type, or each of the types in the operand tuple.* This is a direct consequence of standard tuple subtyping (see formula (5) below) and the homomorphism restriction. We shall call such operators type-covariant.

To define the type transformation associated with an overloaded operator two properties must be stated. First of all, as was mentioned earlier, the input type of the highest instance $L_{\max}$ may be less than the maximum available type. For example, the maximum instance of the operator $>$: $(\tau, \tau) \to bool$ is the one with $\tau = real$ as complex numbers are not comparable. This leads to the requirement that each operator must state its maximum admissible operand types: $(a_1^{\max}, \ldots, a_n^{\max})$, where n is the arity of the operator.

Secondly, the relationship between the input and output types must be stated. According to the above Corollary, the output type is a non-decreasing function of input types. The structure of this function is not arbitrary and is governed by the following

Theorem 1. *For any homomorphically-overloaded n-ary operator $L : (a_1, a_2, \ldots, a_n) \to b$, the output type b can be expressed in the following form*:

$$b(a_1, a_2, \ldots, a_n) = \max(f_1 a_1, f_2 a_2, \ldots, f_n a_n)\ ,$$

where $f_1 \ldots f_n$ are some non-decreasing step functions from types to types defined on the range $a_k \leq a_k^{max}$:

$$f_k x = T_k^{[j_k]} \text{ where } j_k = \min\{i \mid x \leq t_k^{[i]}\} \qquad (1)$$

Here for all k, $T_k^1, \ldots, T_k^\mu$ and $t_k^1, \ldots, t_k^\mu$ are non-decreasing sequences of type constants, μ is the multiplicity of L (i.e. the number of overloadings), and $t_k^\mu = a_k^{\max}$.

Proof is based on Lemma 1.1 and the fact that each operand's type dictates the least admissible overloading separately. If we take the maximum of those overloadings, it will be the least admissible overloading for all the operands' types jointly . Observe that the functions f_k simply select the least overloading $L_i : t_k^{[i]} \to T_k^{[i]}$ whose input type $t_k^{[i]}$ is not less than x. $\triangle$

Although parametrisation (1) is completely general, it is also quite unwieldy making it difficult to find a comprehensive type-inference solution. We have discovered that a less general form of type transformation, namely representing the functions f_k as offset functions $f_k a_k = a_k + m_k$, where m_k is an integer offset, makes homomorphic overloading much more tractable. Here we use the notation $t + n$ with a positive integer n as a shorthand of the nth supertype

[1] ASTL = Array Stream Transforming Language

of t, or $-n$th subtype of t when $n < 0$ [2]. This corresponds to a particular choice of $t_k^{[i]}$ and $T_k^{[i]}$ which limits the variety of homomorphically overloaded operators the type system can represent.

In the remainder of the paper we shall demonstrate that

1. despite the limitation, an important application domain, namely data-parallel array processing, can be supported exclusively by offset signature of this kind;

2. thanks to the particular representation of the functions f_k as offset functions, a computationally-efficient, automatic inference of the exact least types of all variables is possible.

For ease of reference, homomorphically overloaded operators having offset type signatures will be called *offset-homomorphic operators* henceforth.

To conclude this section, let us dwell briefly on the representation of type signatures for offset-homomorphic operators. In the presence of subtyping, one needs to introduce type coercions directly into the concept of type. This can be done at a level of type inference logic, in which case each subexpression would need to be equipped with two type variables: one for the type yielded by it and the other one for the (possibly higher) type expected by the context. Alternatively, coercions could be factored into the operator type signatures themselves. We prefer the latter option, which also means that the type signature ceases to be a type function (yielding the output type based on the input ones) and becomes a type relation. In other words, the type signature indicates which types can be yielded by the expression, or any of its subsequent type coercions, given the input type. Formally this means that the signature has some type variables that are both universally quantified and conditioned by a set of *constraints* — as in constrained type theory presented in [7].

The output part of a type signature will consequently include the relations between the output type and each of the input types. Due to the use of offset functions and the presence of type coercions, each of these relations can be written in the form of inequality as follows (where m is an integer constant and either $t_{1,2}$ is a type variable or a type constant):

$$t_1 \geq t_2 \pm m, \tag{2}$$

We shall refer to Ineq. (2) as the *canonical form* for constraints henceforth. Curiously, due to the multiplicity of potential constraints on a pair of type variables, the canonical form is capable of supporting type *equations* as well: for instance, $t_1 = t_2$ can be represented as $\{t_1 \geq t_2, t_2 \geq t_1\}$. Also note that the canonical form is suitable for representing input constraints, too: the assertion $a \leq a_{\max}$ is equivalent to $a_{\max} \geq a + 0$.

3. CONSTRUCTS OF ASTL

3.1 Arrays and shape

[2]Strictly speaking these are sub- and supertypes within the linearly ordered variety of *admissible* types for each operand of a homomorphically overloaded operator; types may not be linearly ordered in the type system as a whole.

ASTL defines a set of constructs for array manipulation. Unlike most programming languages, it treats all objects as arrays of some rank (i.e. number of dimensions). Objects that have no associated dimensions are called *scalars* and assumed to have the rank 0. The shape of an array is considered to be its dynamic characteristic and is therefore outside the scope of the type system. Conventional binary operations on arrays, such as elementwise addition, multiplication, etc., usually require the operand shapes to be the same. Such a requirement is convenient if conformance can be checked statically in all cases. However, for any comprehensive set of array operations, which include liberal rearrangements of elements and complex selections, array shape quickly becomes untraceable and shape conformance statically unenforceable. ASTL avoids shape-related problems by replacing conformance by the following *intersection rule* : the extent of the result of a binary operation which involves juxtaposition of the operands (such as element-wise addition) is the lesser of the operand extents in each dimension. For example, if A has the shape 100×200 and B 150×70, the result shape of $A \oplus B$, where $\oplus$ is such an operator, is 100×70 .

3.2 Type and subtyping

ASTL objects have two static attributes: element type t and array rank r. These attributes will be referred to as "types" below. Where this may lead to a confusion, the full term "element type" will be used in contrast with "complete type" which includes both the element type and the rank of the object. The set of element types

$$T = \{bool, char, int, real, complex\} \tag{3}$$

has a subtype relation defined on it:

$$bool \subset char \subset int \subset real \subset complex, \tag{4}$$

which is a linear order. Owing to the linear ordering of the types, type increment is well-defined, so we write $t + a$ to denote the a^{th} supertype of t and $t - a$ for its a^{th} subtype, in keeping with the requirements of offset-homomorphism.

The ranks are taken from a range of integer numbers $R = 0..\rho_{max}$with the subtype relation being $\leq$. Conceptually, rank coercion is achieved by infinite replication of the object in additional dimensions. For example, integer scalar 3 can be implicitly coerced to rank 1 by forming an infinite one-dimensional array [3,3,3,...]. In practice, infinite arrays do not remain infinite, since the only reason for coercion is to increase the array rank to match it with the rank of another array for a binary operation. In such a case, the other array will determine the shape of the result according to the intersection rule. For instance, if the above array [3,3,3...] is added to an existing one-dimensional array X elementwise, the result is a new array X' conforming to X, its elements being equal to the corresponding elements of X plus 3. For example, if $X = [2, 4, 6]$, $X + 3 = [5, 7, 9]$.

When x has a type t and a rank r, we write $x : r\#t$.

An operator with more than one operand is assumed to act on the operand tuple. The subtyping relation on 2-tuples is defined in the standard way:

$$(a_1, a_2) \subseteq (b_1, b_2) \equiv a_1 \subseteq b_1 \wedge a_2 \subseteq b_2, \tag{5}$$

and similarly for 3-tuples, etc. A similar rule could be introduced for subtyping the complete types: the element

type and rank must satisfy their subtyping relations simultaneously. However, since all type signatures of ASTL are formulated for element types and ranks separately, the need for subtyping complete types never arises.

3.3 Operators

ASTL operators are families of offset-homomorphic instances where each instance applies to operands of a certain type and produces the result of a certain output type. In keeping with Section 2, their associated type transformations can be described collectively using a type signature in the form **requires** ⇒ **ensures**. The **requires** part defines the maximum operand types for which an operator instance exists. Operands are gathered into a comma-separated list of entities in the form $opd : r\#t.C$, where r is the rank and t the element type variables associated with the operand *opd*, and C is the set of constraints on the element type and rank of *opd* that are imposed by the operation. For example, the **requires** part of a unary operator $\sim$ requiring an operand whose type is not higher than integer and the rank is not higher than scalar is $x : r\#t.\{r \le 0; t \le int\}$. For compactness we shall write constraints in any convenient form as long as the equivalent canonical form exists, for instance, the above $t \le int$ is equivalent to the canonical $int \ge t + 0$.

The **ensures** part defines the constraints on the output type variables: $exp : r\#t.C$, where *exp* is the application of the operator to its operands, r and t are the associated type variables, and C is a set of constraints applied to them as a result of the operator action. For instance, the elementwise comparison $x > y$ of two arrays requires them to be at most real (complex numbers cannot be compared) and ensures that the result is at least *bool* (which is true anyway as *bool* is the bottom type) and has at least the maximum rank of the two operand arrays:

$$\begin{gathered} c \quad x : t_1\#r_1.\{t_1 \le real\}, y : t_2\#r_2.\{t_2 \le real\} \\ \Rightarrow \\ x > y : t\#r.\{t \ge bool; r \ge r_1; r \ge r_2\}\,. \end{gathered}$$

Next we present the type signatures and actions of some ASTL operators. Our goal in this section is to demonstrate that offset homomorphism has a natural meaning for both array computation and geometric manipulations of array elements.

3.3.1 Array constructor

Since the intention of ASTL is to support array processing, it must provide a facility for creating an array object from the function that defines the values of its elements. This is achieved with the help of an array constructor operation, which has the following syntax:

$$array\,(i_1 \mid b_1, ..., i_n \mid b_n)\,exp$$

where $b_1 ... b_n$ are the expressions defining the extents of the array and $i_1, ..., i_n$ are the variables to be used as array indices in the expression *exp*, which defines the content of the corresponding array element. For instance, the array

$$array(x|3, y|4)\ 10 * x + y = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \end{pmatrix}\ .$$

The type signature for the array constructor depends on the value of n which is determined from syntax:

$$\begin{gathered} exp : r\#t.\{r = 0\}, \\ (\forall k,\ 0 < k \le n) b_k : r_k\#t_k.\{r_k = 0;\ t_k \le int\}, \\ i_k : \rho_k\#\tau_k.\{\rho_k = 0;\ \tau_k = int\} \\ \Rightarrow \\ array\,(i_1 \mid b_1, ..., i_n \mid b_n)\,exp : \rho\#\tau.\{\rho \ge n;\ \tau \ge t\} \end{gathered}$$

3.3.2 Recast

An array can be formed by copying and re-packaging some of the elements of another array. Array processing languages traditionally introduce a number of *ad hoc* operations, such as *slice*, *transpose,* etc. By contrast, ASTL offers one general operation, called *recast*, which has the following syntax:

$$exp_0[exp_1, ..., exp_m \leftarrow i_1 \mid b_1, ..., i_n \mid b_n]\ ,$$

where the 0-th expression is the source array, expressions from 1 to m are indices for the source array and $i_1 ... i_n$ are the "input" indices. The bounds $b_1, ..., b_n$ delimit the shape of the resulting array. The values of n and m are determined from syntax. For example, if A denotes the array $array(x|3, y|4)10 * x + y$ used earlier, then

$$A[2 - a, 3 - b \leftarrow a|2, b|2] = \begin{pmatrix} 23 & 22 \\ 13 & 12 \end{pmatrix}$$

If the rank ρ of the source array is less than m then the extra $m - \rho$ indices are ignored (this corresponds to the replication of the argument array in the extra dimensions) and the result rank remains at least[3] n. When $\rho > m$, the extra $\rho - m$ indices are appended to the n input indices on the right, giving the result rank of at least $n + \rho - m$. It is easy to see that both cases are covered by the constraint set $\{r \ge n;\ r \ge \rho + (n - m)\}$, which is in canonical form, giving the signature:

$$\begin{gathered} (\forall k : 1, ..., n) b_k : r_k\#t_k.\{t_k \le int;\ r_k = 0\}, exp_0 : \rho\#\tau \\ \Rightarrow \\ exp_0[exp_1, ..., exp_m \leftarrow i_1 \mid b_1, ..., i_n \mid b_n] : r\#t \\ .\{r \ge n;\ r \ge \rho + n - m;\ t \ge \tau\} \end{gathered}$$

3.3.3 Unary and binary arithmetic and elementary functions

Note that the homomorphism restriction is satisfied by the conventional arithmetic operators $+, -, \times, /$ as far as the element types are concerned, with the proviso that the division operator for integers only yields real results. To preserve the homomorphism, the integer division that discards the remainder must be lexically different (denoted by **div**, for instance). We ignore the fact that it may be possible to add or multiply integer numbers as floating-point numbers correctly, while doing this in the integer type before coercing the result to real would cause an overflow. In such cases it may be desirable to enforce the real type from the start, by applying a type cast operator (see below) to each of the operands. In the same vein, rounding errors can break the homomorphism of comparisons, since two different integer numbers after coercion to the real type may turn out to be rounded to the same floating-point value. Fortunately, this can never be observed in a program, since the typing

[3] We say 'at least' because a higher rank can be expected by the expression context due to the rank subtyping rule.

algorithm described in the next section will assume homomorphism and choose the least type, and it is that type that guarantees the correct answer.

The binary operator $\uparrow$ for raising to a power also satisfies the homomorphism restriction although here the correct definition of instances is slightly trickier. There are three cases of element type: the integer version, usually implemented via repeated multiplication:

$$a : t_1.\{\,\},\ b : t_2.\{t_2 = int\} \Rightarrow a \uparrow_{int} b : t.\{t \geq t_1\};$$

the real version, implemented via real logarithm and requiring a positive base:

$$a : t_1.\{t_1 \leq real\},\ b : t_2.\{t_2 = real\} \Rightarrow a \uparrow_{re} b : t.\{t \geq t_1\}$$

(assuming that a negative base throws an exception, rather than requiring the subtype of nonnegative reals); and the complex version, implemented via complex logarithm,

$$a : t_1.\{\,\},\ b : t_2.\{\,\}$$
$$\Rightarrow$$
$$a \uparrow_{cpx} b : t.\{t = complex\}$$

all of which can be defined jointly as

$$a : t_1.\{t_1 \geq t_2\}, b : t_2.\{\,\} \Rightarrow a \uparrow b : t_3.\{t_3 \geq t_1\}.$$

which represents six instances of the power operator.

The ranks of the operands to a binary operator are required to be equal (up to the rank subtyping), therefore the output rank is greater than or equal to the greater input rank. We list the rest of the signatures of the ASTL computational operators below, without comments; it is easy to see that all of them satisfy the homomorphism restriction.

- $\sqrt{-1}$ imaginary unit
  $\Rightarrow iunit : r\#t.\{t = complex;\ r = 0\}$

- Unary minus
  $a : r_1\#t_1.\{\,\} \Rightarrow -a : r\#t.\{t \geq t_1; t \geq int; r \geq r_1\}$

- Unary negation
  $a : r_1\#t_1.\{t_1 \leq bool\} \Rightarrow \mathbf{not}\, a : r\#t.\{t \geq t_1\}; r \geq r_1$

- Elementary functions:

  $$a : r_1\#t_1.\{t_1 \leq real\} \Rightarrow f(a) : r\#t.\{t \geq real; r \geq r_1\}$$

  for trigonometric functions and

  $$a : r_1\#t_1.\{\,\} \Rightarrow f(a) : r\#t.\{t \geq real;\ t \geq t_1; r \geq r_1\}$$

  for the square root, logarithm and exponential. Note that for correct treatment of exceptions, the type of the operand to logarithm or square root may need to be forced up to complex explicitly if the context suggests the real type, a negative value is possible but the intended interpretation of the operator is at complex type.

- Binary arithmetic $a : r_1\#t_1, b : r_2\#t_2 \Rightarrow a \oplus b : r\#t.\{r \geq r_1;\ r \geq r_2;\ t \geq t_1;\ t \geq t_2;\}$ where $\oplus$ is one of $+$,$-$, or $\times$;

- Comparisons $a : r_1\#t_1.\{t_1 \leq real\}, b : r_2\#t_2.\{t_2 \leq real\} \Rightarrow a \ominus b : r\#t.\{r \geq r_1;\ r \geq r_2\}$, where $\ominus$ is one of $<$,$>$,$=$, $\leq$, $\geq$, or $\neq$.

- If-then-else

  $$c : r_0\#t_0.\{t_0 = bool\}, a : r_1\#t_1, b : r_2\#t_2$$
  $$\Rightarrow$$
  $$\text{if } c \text{ then } a \text{ else } b \text{ endif} : r\#t$$
  $$.\{r \geq r_0; r \geq r_1; r \geq r_2; t \geq t_1; t \geq t_2\}.$$

  This is an operation that uses three array operands, which, after rank coercion and the application of the intersection rule, will have the same shape. The condition and the two alternatives are selected from the corresponding elements of the arrays and a conforming result array is returned.

- Minimum and maximum $a : r_1\#t_1.\{t_1 \leq real\}, b : r_2\#t_2.\{t_2 \leq real\} \Rightarrow a \oplus b : r\#t.\{r \geq r_1;\ r \geq r_2;\ t \geq t_1;\ t \geq t_2;\}$, where $\oplus$ is one of max, min.

- Modulo and integer division $a : r_1\#t_1, b : r_2\#t_2.\{t_1 \leq int;\ t_2 \leq int\} \Rightarrow a \odot b : r\#t.\{r \geq r_1;\ r \geq r_2;\ t \geq int;\}$ where $\odot$ is one of mod, div.

- Binary logic $a : r_1\#t_1.\{t_1 \leq bool\}, b : r_1\#t_2.\{t_2 \leq bool\} \Rightarrow a \wr b : r\#t.\{r \geq r_1;\ r \geq r_2\}$, where $\wr$ is one of $\vee$, $\wedge$

- Exponentiation: $a : t_1\#r_1.\{t_1 \geq t_2\}, b : t_2\#r_2.\{\,\} \Rightarrow a \uparrow b : t_3\#r_3.\{t_3 \geq t_1;\ r \geq r_1;\ r \geq r_2;\}$

- Type cast:

  rounding

  $$a : r_1\#t_1.\{t_1 \leq real\} \Rightarrow$$
  $$toInt(a) : r\#t.\{t = int; r \geq r_1\}\ ,$$

  taking the real/imaginary part

  $$a : r_1\#t \Rightarrow Re/Im(a).r\#t.\{t = real; r \geq r_1\}\ ,$$

  enforcing a type

  $$a : r_1\#t_1.\{t_1 \leq int\} \Rightarrow$$
  $$forceint(a) : r\#t.\{t = int; r \geq r_1\}\ ,$$

  $$a : r_1\#t_1.\{t_1 \leq real\} \Rightarrow$$
  $$forcereal(a) : r\#t.\{t = reali; r \geq r_1\}\ ,$$

  $$a : r_1\#t_1.\{t_1 \leq complex\} \Rightarrow$$
  $$forcecomplex(a) : r\#t.\{t = complex; r \geq r_1\}\ .$$

3.3.4 Concatenation

The operator recast defined earlier is intended for rearrangements of array elements into new arrays. However, recast can use only one array as a source. To be able to use more than one source array, an additional operator is required. Since recast supports arbitrary rearrangements, the new operator need not be flexible with the placement of the elements in the result: a simple concatenation of the operand arrays is sufficient. Accordingly, ASTL has a concatenation operator with the following syntax:

$$exp_1 \sim k \sim exp_2.$$

Concatenation proceeds along the axis indicated by the integer constant $k \in [1, R_{max}]$ placed in the operator symbol, e.g. $a \sim 2 \sim b$ is the concatenation of the arrays a and b along axis 2. If the operands have the rank k or above, the

action of concatenation is straightforward. The extent of the result in the kth dimension is the sum of the operands' extents, while the extents in the rest of the dimensions are governed by the intersection rule. If either operand has a lesser rank, it is coerced to the rank $k-1$ by replication (provided that it is not rank $k-1$ already) and then on to the rank k by assuming the extent 1 in the kth dimension. Then concatenation proceeds as before.

For instance, $1 \sim 2 \sim 2$ is a 2d array having the following values:

$$\begin{pmatrix} 1 & 1 & 1 & \dots \\ 2 & 2 & 2 & \dots \end{pmatrix}$$

The type signature is as follows:

$$a : r_1\#t_1.\{\,\}, b : r_2\#t_2.\{\,\}$$
$$\Rightarrow$$
$$a \sim k \sim b : r\#t.\{t \geq t_1;\ t \geq t_2; r \geq r_1;\ r \geq r_2; r \geq k\}\ .$$

Since nothing happens to the values of the array elements participating in concatenation, the homomorphism in element type is straightforward. The homomorphism in rank is not obvious. In every dimension other than that of concatenation, replication takes place whenever the ranks do not match, hence homomorphism is assured. In the dimension of concatenation homorphism would be broken, but the operation itself is not defined on replicated objects here, as it requires a finite extent of either argument. We conclude that for any valid interpretation of concatenation the homomorphic restriction is satisfied.

3.3.5 Reduction

ASTL follows the standard interpretation of reduction as lowering the rank of an array by inserting a scalar commutative-associative operator between the elements in its first dimension. This is denoted by $\oplus/exp$ where $\oplus$ is the scalar operator in question and exp is an array expression of rank one or above:

$$a : r_1\#t_1 \Rightarrow \oplus/a : t\#r.\{t \geq t_1;\ r \geq r_1 - 1\},$$

where $\oplus : (\tau, \tau) \to \tau$ is one of $+$,$\times$, $\vee$, $\wedge$, $=$, $\neq$ (the last four introducing $t_1 \leq bool$ in the **requires** constraint set).

Reductions can also be made from the max and min operators:

$$a : r_1\#t_1.\{t_1 \leq real\} \Rightarrow \odot/a : t\#r.\{t \geq t_1;\ r \geq r_1 - 1\}$$

where $\odot$ is either min or max.

It is easy to prove that as far as the element type is concerned, if the operator $\oplus$ satisfies the homomorphism restriction then so does the reduction $\oplus/$. With regard to operand rank, the homomorphism restriction is satisfied trivially.

3.3.6 Replication

This operation adds an extra dimension into its operand. The newly added dimension becomes dimension 1, the existing dimension 1 becomes dimension 2, etc. The object is replicated in the new dimension the number of times indicated by the second operand. This is denoted as $A \gg N$. For example,

$$(array(x|3)x) \gg 2 = \begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 2 & 2 \end{pmatrix}\ .$$

Note that replication is *not* equivalent to rank coercion as it adds lower, rather than higher, dimensions and produces the result of a finite shape. Its purpose is to help define arrays of regular structure without resorting to obscure recasts.

The type signature of replication is as follows

$$a : r_1\#t_1.\{\,\}, n : r_2\#t_2.\{t_2 \leq int; r_2 \leq 0\}$$
$$\Rightarrow$$
$$a \gg n : t\#r.\{t \geq t_1;\ r \geq r_1 + 1\}$$

3.3.7 Singleton

A singleton operator increases the rank of an array by 1 without replication. This is a convenient way to create objects such as a single-column matrix, etc., which could then exploit the intersection rule for slicing lower-dimensional subsets from other arrays. There is also an important use of singletons in l-expressions (see section 3.6 below) where replication must be suppressed. A singleton made out of an array A is denoted $!A$, For instance, !5 is a one-dimensional array of size 1 whose only element is the number 5, and !!5 is the same array interpreted as a 1×1 matrix. The singleton operator has a signature similar to that of replication:

$$a : r_1\#t_1.\{\,\} \Rightarrow !a : t\#r.\{t \geq t_1;\ r \geq r_1 + 1\}$$

3.4 Transformer constructs

In the previous sections we have described the expression syntax, semantics and typing of ASTL. Inferring expression types is a trivial problem which can be solved by bottom-up joining of the type signatures associated with the nodes of the expression tree. The process becomes rather more involved when expressions are recursively linked with variables, which is the main evaluation mechanism of declarative languages. In this section we describe the evaluation mechanism of ASTL to prepare the ground for the next section, which describes the type inference method.

The ASTL model of computation is a stream-processing network. The edges of the network represent directed, tuple-valued channels, or *streams*. The nodes have input and/or output streams incident to them and act as *stream transformers*. A stream transformer processes tuples which it reads from one or more input streams and generates new tuples which are sent to the output streams. The tuples of the input streams are combined into a single *input tuple,* which is either locked in place or updated from the input streams at every cycle of the stream transformer depending on an optional *hold* condition. The transformer itself is a recurrence relation between its input and *output* tuples. The output tuple is subjected to optional filtering. At every cycle, if the filter (a Boolean expression) evaluates to *true* the output tuple is released to the output stream(s), otherwise it is discarded. The action of the transformer consists in applying the recurrence relation to the *stream variables* defined in it. The term "recurrence" refers to primitive recursion, i.e. generating new values of the variables based on the current ones. Recurrence relations are common in computational applications, where they are used for defining mesh-based numerical methods, iterative algorithms, etc.

The syntax of the stream transformer is as follows:

$$name\ \langle par_1, ..., par_n \rangle\ out\ in\ body\ ,$$

where the *name* is the transformer identifier, the variables in the angular brackets are non-stream formal parameters, *out* is the output interface and *in* the input interface; the *body*

contains the recurrent relations associated with the transformer. The input interface is omitted if the transformer has no input streams. The output interface has the following syntax:

$$(exp_1, ..., exp_n)\,\textbf{check}\,exp,$$

with a rank-0 Boolean *exp* which may depend on stream variables (see below) and which controls the release of the tuple $(exp_1, ..., exp_n)$ into the output stream. The output expressions can name any variables including the parameters.

The input interface is similar:

$$(var_1, ..., var_n)\,\textbf{hold}\,exp.$$

Here the expression is also a rank-0 boolean, which controls whether the input tuple $(var_1, ..., var_n)$ is held or updated from the input streams. The **hold** expression can name stream variables as well.

ASTL transformers can be either fixed-shape or variable-shape.

3.5 Variable-shape transformer (VST)

A variable-shape transformer body contains any number $m \geq 0$ of stream-variable definitions in the following syntax:

$$var_i \leftarrow exp_{0i},\, exp_i,$$

where $1 \leq i \leq m$, exp_{0i} is the initial value of the *stream variable* var_i and exp_i is the *recurrence step*, i.e. an expression that defines the new value of var_i given the current values of all stream variables. The stream variables are introduced by naming them on the left of $\leftarrow$. They must not clash with any of the variables of the input interface and can be named in the output interface. The transformer produces a stream one stage at a time, by simultaneously evaluating the recurrence steps and associating the results with the corresponding stream variables. For instance, the stream of natural numbers can be programmed as $n \leftarrow 1, n+1$, the stream of Fibonacci numbers as $m \leftarrow 1, k;\ k \leftarrow 1, m+k$, with the actual Fibonacci sequence associated with m, etc. Note that both the initial-values and the step expressions can name formal parameters par_j and variables from the input interface.

Stream-variable definitions generate type constraints from the expressions that they contain; there is an additional constraint stemming from the fact that for each stream-variable definition, the initial value and the result of the recurrence step must be compatible with the type of the stream variable:

$$var : r\#t,\ init : r_1\#t_1,\ step : r_2\#t_2$$
$$\Rightarrow$$
$$var \leftarrow init,\ step : r\#t.\{r_1 \leq r;\ t_1 \leq t;\ r_2 \leq r;\ t_2 \leq t\}$$

Note that the stream variable *var* acquires a contravariant type constraint, i.e. its type is bounded from below.

3.6 Fixed-shape transformer (FST)

A fixed-shape transformer, as the name suggests, defines output streams of fixed-shape array values. Since the shape is fixed, the recurrence step no longer defines the shape of the stream which is now fully determined by the initial values. ASTL puts this circumstance to use by allowing part of the step value to be undefined, taking the undefined part from the corresponding portion of the previous step. Another new feature of fixed-shape transformers is left-expressions in the stream definition. The left-expression is a well-formed expression containing exclusively concatenation, rearrangement and choice operators, i.e. concatenation, recast and **if-then-else**, where the condition part and the recast index expressions are "normal", right-expressions, which have the type signatures defined earlier. The semantics of the left-expression is one of a *target* for the step. The target can be thought of as a reference object which re-directs the values coming from the right-hand side to the receiving array-elements of stream variables; the rest of the elements of the stream variables take their default values. For instance, the step definition $a[i, j \leftarrow i|N-1, j]\ \sim 2 \sim b\ :=\ b \sim 2 \sim a$ (assuming that the initial values of a and b are shaped as $N \times N$ matrices) causes the values of the b elements to be assigned to the corresponding elements of the matrix a except the last row which remains the same as the last step; and at the same time the old values of a are assigned to the corresponding values of the matrix b. The shapes of the target and the right-hand side are subject to the same intersection rule as the operands of a binary elementwise operation.

The syntax of the fixed-shape transformer includes two sections, *initial* and *update*. The *initial* section contains initialisations of stream variables in the form $var = exp$, where *var* is a stream variable and *exp* is the expression defining its initial value. These expressions can only name stream parameters par_j . The *update* section consists of the step declarations in the form $lexp\ :=\ exp$ explained above. The left-expression *lexp* cannot name variables of the input interface or any of the stream parameters, except in conditions and index formulae where they are allowed. The need for these restrictions is straightforward from the semantics of the target.

Let us now analyse the type signatures of the l-operations. Consider the element type first. The target will accept elements of the type that is equal or junior to the least element type of any element of the target. Indeed the right-hand side will yield an object each element of which is of the same type. No element of the target can refer to a stream variable of a lesser type since in that case it would not be possible to assign the corresponding value. This principle must be applied recursively to the l-expression structure and is a manifestation of the same contravariance of type as was observed with VSTs.

Next, looking at rank coercion, an l-expression cannot use replication for its elements as this would result in multiple values of the right-hand side directed to the same target element of the left-hand side, an ambiguous assignment. Consequently l-expressions must not allow rank subtyping if that involves replication of target elements. In the case of concatenation along the kth axis, the operands' rank must not be less than $k-1$; if one operand's rank is k the other one can be k or $k-1$, otherwise the ranks must match. Any other rank combination involves replication. Unfortunately this constraint is not offset-homomorphic because it involves logical condition besides an offset. The solution lies in forbidding the $(k, k-1)$ combination. The singleton operator must be used instead ensuring that the operand ranks are equal in all cases. For example, the r-expression

$$1\ \sim 1 \sim\ array(x|2)\,x + 2$$

is legal and yields the array (1,2,3) but a similar l-expression

$s \sim 1 \sim v$ with a scalar s and a 1d array v causes rank mismatch and hence is not allowed. It should be written as $(!s) \sim 1 \sim v$ on the left-hand side.

To summarise, the signature of l-concatenation is as follows:

$$a : r_1\#t_1.\{r_1 \geq k\}, b : r_2\#t_2.\{r_2 \geq k\}$$
$$\Rightarrow$$
$$a \sim k \sim b : r\#t.\{t \leq t_1;\, t \leq t_2; r = r_1;\, r = r_2\}\ .$$

L-recast and l-if-then-else operators cause no difficulty in arriving at the following fixed-rank, contravariant signatures:

$$(\forall k : 1, ..., n) b_k : r_k\#t_k.\{t_k \leq int;\, r_k = 0\},$$
$$exp_0 : \rho\#\tau.\{\,\}$$
$$\Rightarrow$$
$$exp_0[exp_1, ..., exp_m \leftarrow i_1 \mid b_1, ..., i_n \mid b_n] : r\#t$$
$$.\{r = \rho + n - m;\, t \leq \tau\}$$

L-if-then-else

$$c : r_0\#t_0.\{t_0 = bool\}, a : r_1\#t_1, b : r_2\#t_2$$
$$\Rightarrow$$
$$\text{if } c \text{ then } a \text{ else } b \text{ endif} :$$
$$r\#t.\{r = r_0 = r_1 = r_2; t \leq t_1; t \leq t_2\}$$

Note that parts of the last two signatures have covariant constraints where values rather than targets are involved. The formal basis for contravariance of the target types is not very obvious. A target can be thought of as a function which receives the value from the right-hand side and produces the new stage value of the corresponding stream variable. A target-to-target coercion is in this sense a coercion of an arrow to the same arrow except for a different argument type. It is well known from the theory of subtyping [2] that an arrow $x \to y$ is contravariant with respect to x and covariant with respect to y so $x \to y$ is a subtype of $x' \to y'$ iff $x' \subseteq x$ and $y \subseteq y'$, which agrees with our treatment of targets.

From the implementation point of view, a target of type t can be interpreted as "reference-to t". However, since geometric transformations are semantically neutral to the nature of the elements, the difference between an ordinary element type t and the reference type $t' = ref(t)$ can be addressed in implementation and ignored in type inference.

Each definition in the *initial* section produces the following constraint on the stream variable named in it:

$$exp : r_1\#t_1.\{\} \Rightarrow var = exp\ :\ r\#t.\{r \geq r_1;\, t \geq t_1\}\ .$$

A definition in the *update* section is treated as follows:

$$lexp : r_l\#t_l,\ rexp : r_r\#t_r$$
$$\Rightarrow$$
$$lexp := rexp\ \ :\ .\{r_r \leq r_l;\, t_r \leq t_l\}\ .$$

where the construct as a whole does not return any value and so has no type, but still has constraints on the types of expressions that occur in it. Also note, that formally := is a binary element-wise operator and hence the intersection rule is used in determining the shapes of the right- and left-hand-sides involved in the actual update.

4. TYPE INFERENCE

The type rules described in earlier sections decorate the abstract syntax tree of an ASTL program with type attributes (t_k and r_k) connected by type constraints. If a tree node represents an occurrence of a variable then the type attributes are associated with the variable and remain the same for every occurrence of that variable within its scope. Otherwise, the node represents a (sub)expression, to which a fresh set of attributes is attached.

4.1 Step 1. Constraint set expansion

Each of the type constraints gathered from branches of the syntax tree is in canonical form (2) and defines a *primary* type constraint in the program. The primary constraints also include the constraints on constant types and ranks which can be written in canonical form as follows:

$$integer \geq boolean + 1$$
$$boolean \geq integer - 1$$
$$real \geq integer + 1$$
$$integer \geq real - 1$$
$$\dots$$

with the bottom and top elements of the type system satisfying additional consistency constraints:

$$(\forall t \in V_t)(t \geq boolean) \wedge (complex \geq t)\,,$$

where V_t is the set of all type variables associated with the syntax tree. Similar constraints are introduced for constant ranks:

$$\rho_1 \geq \rho_0 + 1$$
$$\rho_0 \geq \rho_1 - 1$$
$$\rho_2 \geq \rho_1 + 1\ .$$
$$\rho_1 \geq \rho_2 - 1$$
$$\dots$$

and on their bottom and top elements:

$$(\forall r \in V_r)(r \geq \rho_0) \wedge (\rho_{\text{top}} \geq r)\,,$$

where V_r is the set of rank variables. Thus variable and constant type attributes are treated uniformly.

By combining available constraints pairwise where two constraints have one or two common variables, new constraints in canonical form can be produced, which we shall call *secondary*. Denote the set of all primary and secondary constraints of the program as $\mathbb{C}$. Two rules are proposed: a **chain** rule,

$$\frac{\tau_1 \geq \tau_2 + a\ \in \mathbb{C} \quad \tau_2 \geq \tau_3 + b\ \in \mathbb{C}}{\mathbb{C} \to \mathbb{C} \cup \{\tau_1 \geq \tau_3 + (a + b)\}} \qquad (\textbf{chain})$$

and a **cull** rule

$$\frac{\tau_1 \geq \tau_2 + a\ \in \mathbb{C} \quad \tau_1 \geq \tau_2 + b\ \in \mathbb{C},\ b < a}{\mathbb{C} \to \mathbb{C} \setminus \{\tau_1 \geq \tau_2 + b\}}\,. \qquad (\textbf{cull})$$

Here and below we use meta-variables τ_i that range over type and rank variables of the program. It is easy to see, that (**chain**) and (**cull**) are the only ways of deducing new information from two inequalities in canonical form. To obtain *all* relationships between the types, chaining and culling must be applied repeatedly until a fixed point is reached, and it will be reached eventually (see Theorem 2 below). To

avoid spurious cycles where a constraint is added by chaining and then immediately culled out, we modify the chain rule thus:

$$\frac{\begin{array}{c}\tau_1 \geq \tau_2 + a,\ a > -\infty\ \in \mathbb{C} \\ \tau_2 \geq \tau_3 + b,\ b > -\infty\ \in \mathbb{C} \\ \tau_1 \geq \tau_3 + c\ \in \mathbb{C},\ c < a + b\end{array}}{\mathbb{C} \to (\mathbb{C} \setminus \{\tau_1 \geq \tau_3 + c\}) \cup \{\tau_1 \geq \tau_3 + (a+b)\}}\,, \qquad (\textbf{chain'})$$

and assume that for each pair of type variables t_i,t_j the relation $t_i \geq t_j - \infty$ is added to the set of primary constraints, and then culling is applied as many times as possible before the rest of the inference process takes place. Note that the constant offsets a and b, while required to be finite, can still be negative.

Next introduce an **inconsistency** rule:

$$\frac{\tau \geq \tau + a\ \in \mathbb{C},\ a > 0}{\mathbb{C} \text{ is inconsistent}} \qquad (\textbf{incons})$$

and a **tautology** rule

$$\frac{\tau \geq \tau + a\ \in \mathbb{C},\ a \leq 0}{\mathbb{C} \to \mathbb{C} \cup \{\tau \geq \tau\}}\,, \qquad (\textbf{taut})$$

both of which have the obvious meaning.

The above rules act as inference rules for deducing secondary constraints from the primary ones. They are sound by construction, as the derived inequalities are logically implied by the premises. It is unclear, however, whether the proposed rules are complete, i.e. whether any inequality whose truth is implied by the primary constraint set can be proven by a finite application of the rules.

The following theorem gives an affirmative answer to the question about completeness. First we need to introduce some terminology. A *chain* between τ_1 and τ_n is a finite set of constraints in the form:

$$\begin{array}{c}\tau_1 \geq \tau_2 + s_1 \\ \tau_2 \geq \tau_3 + s_2 \\ \dots \\ \tau_{n-1} \geq \tau_n + s_{n-1}\end{array}$$

The chain is called *open* if $\tau_n \neq \tau_1$ otherwise it is *closed*. For any chain between τ_1 and τ_n, we derive the secondary constraint $\tau_1 \geq \tau_n + \sigma$, where $\sigma = \sum_{k=1}^{n-1} s_k$ is called the *strength* of the chain. Now we are ready to state

Lemma 2.1. *If a constraint set contains no closed chains of positive strength, for every type variable t there exists a chain with $\tau_1 = t$ such that no other chain with $\tau_1 = t$ is stronger than it.*

Proof. Consider an arbitrary chain between $\tau_1 = t$ and τ_n. If, for some positive $i < j \leq n$, $\tau_i = \tau_j$, then the chain contains a closed subchain between τ_i and τ_j which, by the premise of the lemma, has a non-positive strength. So it can be removed from the chain without decreasing its strength. We can therefore discard all such chains, and since only a finite number of chains is left, there exists one (or more) with the maximum strength. △

Lemma 2.2. *A constraint set is satisfiable iff it contains no closed chains of positive strength.*

Proof. The "only if" part is proven immediately by observing that a closed chain of positive strength implies $\tau_1 \geq \tau_1 + s$ with $s > 0$ which is a contradiction. The "if" part is proven by showing that in the absence of a closed chain of positive strength, there exists an assignment of types to the type variables that satisfies all primary constraints.

Consider the following type assignment:

$$t_i = \bot + S_i\,, \qquad (6)$$

where S_i is the strength of the maximum-strength chain beginning at t_i, which exists according to Lemma 2.1, i enumerates all type variables, and $\bot$ is the bottom type $(\forall i) t_i \geq \bot$ of the type system. Obviously $(\forall i) S_i \geq 0$.

Eq. (6) satisfies all primary constraints. Indeed, for any primary constraint

$$t_i \geq t_j + c_{ij}$$

observe that it forms a chain from t_i if used as a prefix to the strongest chain from t_j. So the right-hand side represents the bottom type offset by the strength of a chain from t_i, while the left-hand side represents the bottom type offset by the strength of the strongest such chain. This means that the inequality holds and since i and j are arbitrary indices, the whole set of primary constraints is satisfied. △

Theorem 2 (Completeness). *If an inequality $t_i \geq t_j + c$ is implied by a satisfiable primary-constraint set, it can be proven by a finite number of applications of the inference rules.*

Proof. By the premise the constraint set $\mathbb{C} \cup \{t_j \geq t_i - c + 1\}$, where $\mathbb{C}$ is the primary constraint set, is unsatisfiable. By Lemma 2.2, this means that a closed, positive-strength chain exists in that set. Since it does not exist in $\mathbb{C}$, we conclude that the chain includes the added inequality. Split that chain into the open chain between t_i and t_j and the added inequality[4]. Apply the (**chain**) rule to the open chain repeatedly to derive $t_i \geq t_j + d$, with some d. From the positive strength of the closed chain we have $d - c + 1 > 0$ which means $d \geq c$ and so $t_i \geq t_j + c$ is proven.△

It should be noted that the fact of completeness by itself does not provide an algorithm to find all the consequences of the primary constraints nor a method of selecting appropriate inference rules for proving a constraint to be a logical consequence of the primary set. It is for this reason that additional rules have been introduced even though Theorem 2 indicates that (**chain**) is the only rule required for completeness. Their equivalence to (**chain**) is due to the following

Theorem 3. *For every derivation that uses (***chain'***) and (***taut***) to prove a secondary constraint, there exists a derivation that uses exclusively the rule (***chain***) and proves the same constraint.*

Proof. Observe that the difference between applying (**chain'**) and (**chain**) is that the former additionally removes from the set a constraint involving the same pair of variables that has been used in the proof already. Such removal is irrelevant as we are free not to repeat the variables involved with the proof for reasons disclosed in the proof of Lemma 2.1. An application of (**taut**) adds a constraint between a type variable and itself, which can be left out of any derivation for the same reasons. △

The next theorem shows the utility of the full set of inference rules.

Theorem 4. *The constraint set reaches either a fixed point or inconsistency after a finite number of applications of (***chain'***) (***taut***) and (***incons***).*

[4]The open chain can be empty if the added inequality is self-contradicting, e.g. $t \geq t + 1$, in which case the proof involves zero applications of the inference rules.

Proof. The process of expanding the constraint set can be re-formulated in the language of graph theory as follows. Place type variables at different vertices of a weighted directed graph that has no edges initially. For every constraint in canonical form $\tau_1 \geq \tau_2 + a$ add the edge (τ_1, τ_2) with the weight $(-a)$ (which we shall call the "distance" between τ_1 and τ_2 henceforth). Then an application of (**chain'**) has either no effect or it replaces the edge between vertices t_1 and t_3 by one whose distance corresponds to the shorter route from t_1 to t_3 via t_2 . Repeated applications will have the effect of inserting minimum distance shortcuts between vertices. Observe that if the distance between any t and itself becomes negative in the course of expansion, $\mathbb{C}$ is found inconsistent straight away. On the other hand, if that distance is positive, an application of (**taut**) makes it equal to zero and places a 0-length loop on a vertex (which is effective only once). We conclude that the graph in question has only zero-weighted loops and no cycles of negative distance. Under such circumstances, the application of (**chain'**) to non-distinct vertices has no effect. When all three vertices are different, the rule shortens an edge by giving it the distance of a (shorter) path between the vertices it is incident to. It is easy to see that as the number of paths between any two vertices is finite, there can only be a finite number of applications of the rule — hence a fixed point is reached or an inconsistency is detected. $\triangle$

In practice an efficient minimum-distance algorithm [5] can be used to expand the constraint set up to its fixed point. The result of the constraint-set expansion is a matrix $W = w_{ij}$ such that for all i, j $(-w_{ij})$ is the minimum distance between the types t_i and t_j according to the program tree. This matrix is, in fact, the direct sum of two matrices: one corresponding to element type and one to rank, since these attributes do not mix in any constraints and since none of the rules causes them to become mixed either. For unrelated types the corresponding matrix element contains $-\infty$.

4.2 Step 2. Minimisation

The task of type inference, understood as a constraint-satisfaction problem, is to assign the least type value for every type variable given the constraints. We are interested in the least admissible types of objects because they correspond to the least instances of overloaded operators applied to these objects. We assume that the higher the type the higher the computational cost incurred by the operator. This assumption is justified for the numerical operators acting on arrays in any reasonable implementation; it is also likely that operators of a different nature would have an associated cost which increases with type.

The type minimisation algorithm must take into account the fact that some of the type variables are external to the program (i.e., those associated with input streams and parameters) and so cannot be mimimised without the knowledge of the external constraints. Moreover, the minimum of each internal type variable may depend on the values of the external ones. In a distributed stream processing environment, the external types of a stream transformer may themselves be dependent on its output types when, for example, an output stream of transformer X is fed into transformer Y and the output of Y is part of the input to X. This problem will be discussed later; for now let us assume that the external types are known.

The minimisation process is performed for element types and ranks separately and identically since they are not mixed in any of the constraints and since the inference logic is insensitive to the nature of type variables. For the avoidance of confusion, in the rest of the section we shall focus exclusively on element types.

Let us partition the set of all type variables $\mathbb{V} = \{t_i \mid 0 \leq i < n\}$ into the set of external variables $\mathbb{E} = \{t_i \mid 0 \leq i < m\}$ with the bottom element *boolean* included as t_0, and the set of internal variables $\mathbb{I} = \{t_i \mid m \leq i < n\}$ so that each type variable has an index within the appropriate range. Next we recall the fixed-point constraint matrix $W = w_{ij}$, which was introduced at the end of the previous section, so that each constraint in the expanded set is represented as

$$t_i \geq t_j + w_{ij} \,.$$

Since external variables do not occur on the left-hand side of the stream transformer construct (where they would be expected to have sufficiently high types to be able to receive the values of the right-hand side), their associated *type* variables do not occur on the left-hand side of any constraints, i.e. $(\forall k < m) w_{kj} = -\infty$ and hence need not be minimised. In fact, the minimum of all these types *due to a program where they are external* is obviously the bottom type of the type lattice.

The solution of the minimisation problem is given by the following **Theorem 5**. *For any values of the external type variables that satisfy the constraints between themselves*

$$t_i \geq t_j + w_{ij} \text{ where } t_{i,j} \in \mathbb{E}$$

the type assignment for the internal variables $t_i \in \mathbb{I}$

$$t_i = \max_{0 \leq j < m} t_j + w_{ij} \tag{7}$$

is the finite, least-type solution of the constraint satisfaction problem.

Proof Substitute the solution into a primary constraint between any two internal type variables t_i and $t_{i'}$ $(m \leq i < n$ and $m \leq i' < n)$:

$$t_i \geq t_{i'} + w_{ii'} \,.$$

Since addition is distributive over maximum, this is equivalent to

$$\max_{0 \leq j < m} t_j + w_{ij} \geq \max_{0 \leq j < m} t_j + w_{ii'} + w_{i'j} \,.$$

Remember that w is a fixed-point matrix, so

$$w_{ii'} + w_{i'j} \geq w_{ij} \,,$$

which means that every term under maximisation on the right-hand side does not exceed the corresponding term on the left-hand side and so the constraint holds. Next observe that the solution (7) satisfies the constraints between the internal and the external variables, i.e. $t_i \geq t_j + w_{ij}$, where $0 \leq j < m \leq i < n$, and finally that it is the least such solution.

The last issue is the finiteness of the type assignment. Infinite negative offsets in the constraints were introduced as a tool which enabled us to ignore the fact that not all pairs of types are mutually constrained. However, we must ensure that any type *assignment* uses only finite offsets from any types that serve as a basis. Two factors guarantee that.

[5] for example, the Floyd-Warshall algorithm, which computes the minimum distance between all pairs of vertices in a directed graph in cubic time.

Firstly, one of the "external" type variables is the bottom type of the type hierarchy, t_0. Since for all i, $t_i \geq t_0$ and so $w_{i0} \geq 0$, at least one term in Eq. (7) is bounded from below. Secondly, if an infinite negative offset occurs in another term, thanks to the max operation that term will have no effect in the presence of a finite term. $\triangle$

Corollary *The least type of an internal type variable is a nondecreasing function of the types of any external type variables.* This follows from the structure of Eq. (7) and the fact that maximum is monotonic with respect to each of its arguments.

4.3 Step 3. Constraining external type variables

All external types $t_j \in \mathbb{E}$ must satisfy the trivial constraints $t_j \geq t_0$ which cannot be strengthened for reasons mentioned earlier; so $w_{j0} = 0$ for all $j < m$. More importantly, however, in the process of expanding the constraint set, secondary constraints in the form

$$t_0 \geq t_j - c_j$$

where $0 \leq j < m$ and $c_j = -w_{0j}$, may appear (note that $c_j \geq 0$ since t_0 is the bottom type). As they are equivalent to $t_j \leq t_0 + c_j$, they set the maximum admissible type of any external t_j to $t_0 + c_j$.

To conclude this section, it is interesting to note that the type minimisation problem can be regarded as an integer programming [9] problem with the objective function

$$g = \sum_{i=m}^{n-1} t_i - t_0 \ ,$$

the variables t_j, $0 \leq j < m$, acting as free parameters and the primary constraints defining the optimisation domain. Indeed, as we must minimise all internal types given the constraints, and since all types are bounded from below by the bottom type with which they are comparable as supertypes, the above sum reaches the minimum only when each of the variables occurring in it is minimised. Although the toolkit of integer programming is too general for our purposes (and we already have an efficient constraint satisfation algorithm with minimisation), this angle on the problem may prove important for non-offset situations.

5. DISTRIBUTED TYPE INFERENCE

When stream transformers are deployed on different hosts included into a Computational Grid, the streams become real network connections. Leaving aside the problem of configuring ASTL network, we remark that any configuration of streams must be consistent with the external types of the stream transformers. The procedure outlined in the previous section obtains a type relation for every transformer whereby the type of the variables in the output interface is dependent on the type of the variables in the input interface and the type of the transformer parameters. The parameters are set directly by the Grid infrastructure, whereas the inputs and outputs are matched with the outputs and inputs of other transformers, respectively. However, as mentioned earlier, a transformer's external (i.e. input) types can no longer be considered as given.

The following distributed type-reconciliation procedure is proposed. Each transformer approximates its input type variables as the bottom type/rank and calculates the type of the output variables using Eq. (7). These output types are passed along the data links to the receiving transformer in a special message. When a transformer receives such a message, it treats the received types as a new approximation of the input types. It then generates a new approximation of the output types to be sent to the corresponding inputs, etc.

According to the corollary to Theorem 5, since the input types increase with each approximation, the output types stay the same or increase as well. If the output types of the transformer remain the same after an approximation, a termination point is reached: no more messages will be sent by this transformer. It is easy to see that after a finite number of messages have been passed between the transformers, each of them either reaches a termination point or receives types that violate the maximum type constraints as defined in section 4.3. In either case no more messages will be sent.

All transformers in a transformer 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.

Conclusions

We have shown that offset-homomorphic overloading is a natural mechanism for describing a variety of array operations in a declarative language. All basic constructs for array computation and geometric manipulation of arrays have been shown to satisfy the offset-homomorphism restriction. We have proposed, and proven the correctness of, a type inference method which deduces the type attributes of a stream transformer from the external type information. A procedure of reconciling types in a distributed stream-processing network has been delineated and will be published separately.

Future work will include optimising the shortest-path algorithm towards the specific structure of the type-dependency graph induced by a syntax tree; for instance, the fact that there are many type variables that are localised each in a small portion of the tree needs to be taken into account as it may improve the performance of the type-inference method. It is perhaps even more important to address the problem of error reporting. Since type errors may result from violating secondary constraints, it is not straightforward (if at all possible) to relate them back to primary constraints and ultimately to specific occurrences of operators in expressions. This problem is common to most type inference situations; an inference algorithm usually points to a mismatch between type signatures and not to the offending operator directly. While inference by unification makes it relatively easy to find which of the signatures that fail to unify is to blame, in our case there can be a whole trail of chained constraints leading to a contradiction. Future work will attempt to suggest useful heuristics in the framework of the ASTL language. We expect the problem of error reporting in a distributed environment to prove especially interesting.

The author wishes to thank the anonymous referee for the encouraging remarks and Prof Chrystopher Nehaniv for his attention to this work and a number of useful discussions.

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. I. Foster, C. Kesselman and S Tuecke. The Anatomy of the Grid. Enabling Scalable Virtual Organisations. Intl. J. Supercomputer Applications, 2001. Also see citeseer.nj.nec.com/492895.html.

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. Three approaches to type structure. In: *TAPSOFT proceedings, LNCS* vol 185, pp.97-138, 1985.

7. 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.

8. 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.

9. A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1998.