Geometry Aware Types For Reference Frames (Gator)

Dietrich Geisler

1 Syntax

Literals in our language can be scalar numbers, n-tuples, or matrices:

$$
\begin{aligned}
s &\in \mathbb{R} \\
v_n &::= [s_1, s_2, \ldots, s_n] \\
m_{n_1 \times n_2} &::= [[s_{11}, s_{12}, \ldots, s_{1n_2}], \ldots, [s_{n_1 1}, s_{n_1 2}, \ldots, s_{n_1 n_2}]]
\end{aligned}
$$

Expressions can be variables, literals, or unary/binary operations:

$$
\begin{aligned}
&x \in \text{variables} \\
&c ::= s \mid v_n \mid m_{n_1 \times n_2} \\
&e ::= c \mid \tau\ x = e \mid x = e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid
\end{aligned}
$$

The foundation of our type system is a *geometric type*, which can only be applied to n-tuples. A geometric type is intended for describing the context needed to represent that n-tuple geometrically and to ensure that geometric representations are not inadvertently mixed.

A geometric type is built upon three components: a reference frame, a geometric object, and a coordinate system. The reference frame of a geometric type gives the origin and orientation by which an inhabiting n-tuple can be interpreted. A reference frame also gives the dimension of the n-tuple, which restricts the size of inhabiting members, through defining a top and bottom type of each dimension (similar to how Lathe defined tag types). The geometric object describes what the n-tuple is, such as a point or direction. Finally, the coordinate system describes how operations on the inhabiting n-tuple, such as rotation and translation, behave. Each of these components is itself a type, but they generally cannot be inhabited by any expressions in our language. The one exception to this is the scalar type, which can be inhabited by exactly real numbers.

The complete type system can be summarized as containing these geometric types, their component types, function types, and built-in scalar and unit types:

$F \in$ frames
$O \in$ objects
$C \in$ coordinates
$n \in \mathbb{N}$
$\phi ::= F \mid \bot_{\phi,n} \mid \top_{\phi,n}$
$\omega ::= O \mid \bot_{\omega} \mid \top_{\omega}$
$\chi ::= C \mid \bot_{\chi} \mid \top_{\chi}$
$\gamma ::= \chi \sim \omega \sim \phi \mid \bot_n \mid \top_n \mid \ \mid \mathsf{scalar}$
$\tau ::= \mathsf{unit} \mid \gamma \mid \gamma_1 \to \gamma_2$

Scalars are special geometric types that are the only geometric type not represented by n-tuples. Function types $\gamma_1 \to \gamma_2$ operate the same as in Lathe. The bottom type $\bot$ of each geometric type component allows us to describe literals and provide a mechanism for 'disregarding' select irrelevant components when declaring a geometric type. The top type $\top$ for each component is only included to allow correct type declarations of matrix literals. Finally, we denote any member of the set of operations ops in Gator consisting of the binary operations $\{+, -, *\}$ as $\odot$.

2 Example Contexts

There are two primary contexts that give meaning to the geometric objects and coordinate systems introduced by Gator:

- Ω is a map from triples – with elements $\gamma, \gamma, \odot$ – to τ. Ω is assumed to only act on the geometric object ω of all non-scalar types.
- $\mathbf{X}$ is a map from triples of elements $\gamma, \gamma, \odot$ to functions, providing definitions for actual operation behavior under that coordinate system. How functions are defined is omitted from this formalism, but it is worth noting that functions mapped to by $\mathbf{X}$ are expected to take in two arguments of the correct type for unary and binary operations respectively.

These definitions are non-trivial, so it is useful to evaluate some examples of how defining a type in each context gives useful geometric behavior. While each of these examples act on geometric types, we will only specify one component, assuming that the other components are unified in the return value (potentially to the top type) via subsumption described in section 4.1.

1. vector geometric object
    - $\Omega[(\mathsf{vector}, \mathsf{vector}, +)] \quad = \mathsf{vector}$
    - $\Omega[(\mathsf{vector}, \mathsf{vector})] \quad = \mathsf{vector}$
    - $\Omega[(\mathsf{vector}, \mathsf{scalar}, *)] \quad = \mathsf{vector}$

2. position geometric object

   - $\Omega[(\mathsf{position}, \mathsf{position}, -)] = \mathsf{vector}$
   - $\Omega[(\mathsf{position}, \mathsf{vector}, +)] = \mathsf{position}$

3. cartesian coordinate system

   - $\mathbf{X}[(\mathsf{cartesian}, \mathsf{cartesian}, +)] = \lambda xy.x + y$
   - $\mathbf{X}[(\mathsf{cartesian}, \mathsf{cartesian}, -)] = \lambda xy.x - y$
   - $\mathbf{X}[(\mathsf{cartesian}, \mathsf{scalar}, *)] = \lambda xs.x * s$

4. polar coordinate system (polar)

   - $\mathbf{X}[(\mathsf{polar}, \mathsf{scalar}, *)] = \lambda xs.[x[0] * s, x[1]]$
   - $\mathbf{X}[(\mathsf{polar}, \mathsf{polar}, +)] =$ ommited for simplicity

Note that we assume the usual notion of vector addition when using the + symbol in a function definition. Note also that the definition of polar coordinate operations relies on inhabitants of polar coordinate being 2-dimensional. We will not be discussing this requirement as $\mathbf{X}$ is not type-checked in this formalism; however, dimension requirements are always constants or simple additive expressions in our experience.

3 Subtype Ordering

Subtype ordering works precisely the same as in Lathe, with the following changes.

First, the context Δ becomes a map from reference frames to reference frames.

Second, we define the following rules for ordering geometric types γ_1, γ_2 on each component:

$$\frac{\gamma_1 = \chi_1, \omega, \phi \qquad \gamma_2 = \chi_2, \omega, \phi \qquad \chi_1 \leq \chi_2}{\gamma_1 \leq \gamma_2}$$

$$\frac{\gamma_1 = \chi, \omega_1, \phi \qquad \gamma_2 = \chi, \omega_2, \phi \qquad \omega_1 \leq \omega_2}{\gamma_1 \leq \gamma_2}$$

$$\frac{\gamma_1 = \chi, \omega, \phi_1 \qquad \gamma_2 = \chi, \omega, \phi_2 \qquad \phi_1 \leq \phi_2}{\gamma_1 \leq \gamma_2}$$

Note that this rule, along with our Lathe construction of delta ordering on reference frames, implies that geometric types form lattices under a fixed coordinate system, geometric object, and dimension. We can use this lattice construction to recover n-tuple dimension from the type in exactly the same method as Lathe, noting that both coordinate system and geometric object can be erased when recovering dimension.

Third, we require that each component bottom and top type works as such:

$$\forall c \in \chi, \bot_\chi \leq c, c \leq \top_\chi$$
$$\forall o \in \omega, \bot_\omega \leq o, o \leq \top_\omega$$

4 Static Semantics

Our typing judgment is a map from an expression under the contexts Γ, Ω to the type of that expression and updated variable context Γ'.

4.1 Subsumption

Subsumption works as in Lathe

4.2 Constants and Variable Declarations

Declaring variables and literals works the same as in Lathe. Note that vector and matrix literal types remain the same despite the changes to $\bot_n$ and $\top_n$, as the ordering behavior of these types is ultimately unchanged in Gator.

4.3 Binary Operations

Binary operation typechecking differs primarily from Lathe in that for some operations we need to verify with Ω that the operation should typecheck. We also provide some special rules for matrix reasoning.

For core operations $\odot$, we have the following rule:

$$\frac{\dfrac{\Gamma, \Omega \vdash e_1 : \tau_1, \Gamma' \qquad \Gamma', \Omega \vdash e_2 : \tau_2, \Gamma''}{\tau_1 = \gamma_1 \qquad \tau_2 = \gamma_2 \qquad \Omega[(\gamma_1, \gamma_2, \odot)] = \tau}}{\Gamma, \Omega \vdash e_1 \odot e_2 : \tau_3, \Gamma''}$$

Note that scalar operations are assumed be included in Ω.

Matrix operations work a bit differently. Since matrices are just functions, we allow matrix multiplication to still apply to any vector of the appropriate type. Additionally, the linear properties of a matrix only apply when they apply to the *domain* of the matrix function.

$$\frac{\Gamma, \Omega \vdash e_1 : \gamma_1 \to \gamma_2, \Gamma' \qquad \Gamma', \Omega \vdash e_2 : \gamma_1, \Gamma''}{\Gamma, \Omega \vdash e_1 * e_2 : \gamma_2, \Gamma''}$$

$$\frac{\Gamma, \Omega \vdash e_1 : \gamma_1 \to \gamma_2, \Gamma' \qquad \Gamma', \Omega \vdash e_2 : \gamma_1 \to \gamma_2, \Gamma'' \qquad \Omega[(\gamma_2, \gamma_2, +)] = \gamma_3}{\Gamma, \Omega \vdash e_1 + e_2 : \gamma_1 \to \gamma_3, \Gamma''}$$

$$\frac{\Gamma, \Omega \vdash e_1 : \gamma_1 \to \gamma_2, \Gamma' \qquad \Gamma', \Omega \vdash e_2 : \gamma_1 \to \gamma_2, \Gamma'' \qquad \Omega[(\gamma_2, \gamma_2, -)] = \gamma_3}{\Gamma, \Omega \vdash e_1 - e_2 : \gamma_1 \to \gamma_3, \Gamma''}$$

$$\frac{\Gamma, \Omega \vdash e_1 : \gamma_1 \to \gamma_2, \Gamma' \qquad \Gamma', \Omega \vdash e_2 : \mathsf{scalar}, \Gamma'' \qquad \Omega[(\gamma_2, \mathsf{scalar}, *)] = \gamma_3}{\Gamma, \Omega \vdash e_1 * e_2 : \gamma_1 \to \gamma_3, \Gamma''}$$

5 Dynamic Semantics

Operational semantics (or, rather, compilation semantics) require additional structure beyond that provided by Lathe. We start by defining a context to reason about operation behavior.

To recover the correct operational semantics, we now need to know the type of an expression during compilation. To do this, we introduce the context **T**, which provides the type of an expression; the mechanism for **T** can be built-in to our typing judgment, a step which we will omit for simplicity.

The operational semantics of Gator map, in a single step, from an expression and contexts **X** and **T** to a constant and a state σ. σ itself is, as usual, a map from variable names to constants. The only semantics of interest in this language are the translation of vector and matrix operations.

5.1 Mathematical Operations

I'm going to pass on formalizing this for now – I'm not sure whether I want to define semantics directly or by compilation yet, so I'll come back to this once I have a better sense of when replacing operations matters.