Type Inference Rules For Container Types in CCL

[*]The paper is written under the guidance of Dr. Alan K. Zaring (akzaring@luther.edu).

David Oniani
Luther College
oniada01@luther.edu

December 12, 2019

Abstract

We present the type inference rules introducing the notion of container types in the CCL programming language. This redesign required a number of substantial changes to the major aspects of the type system and the syntax of the language. Two new types $\mathcal{Box}$ and IBox were introduced. In addition, two new operators <:: and :: were designed for dealing with the new types.

The $\mathcal{Box}$ and IBox Types

CCL is, by and large, value-based language. This makes it rather difficult to have types more complex than what already comprises CCL. Besides, this is a major obstacle for the implementation of heterogeneous types such as heterogeneous ordered pair, heterogeneous vector types, etc. One possible solution to this problem is introducing the notion of container types.

Container types are boxes which are either mutable or immutable. They wrap the values allowing the composition of compound types by bringing the higher levels of abstraction. Mutable and immutable types are represented by types $\mathcal{Box}$ and IBox respectively. For some type $\mathcal{T}$, the result of the wrapping is either $\mathcal{Box}(\mathcal{T})$ or $\mathit{IBox}(\mathcal{T})$.

The <:: Relation

For expressing relationships between container types, we introduce a new <:: operator. The sole purpose of this operator is to specify the subtype relation between two container types. As an example, $\mathcal{T} <:: \mathcal{U}$ is read as "container type $\mathcal{T}$ is a subtype of the container type $\mathcal{U}$." We proceed by introducing rules based on the new operator.

$$\frac{}{\mathcal{Box}(\mathcal{Triv}) <:: \mathcal{Box}(\mathcal{Triv})} \tag{1}$$

$$\frac{}{\mathcal{Box}(\mathit{Int}) <:: \mathcal{Box}(\mathit{Int})} \tag{2}$$

Rules 1 and 2 specify the reflexive property of the operator on types $\mathcal{Triv}$ and Int.

$$\frac{\mathcal{T} <:: \mathcal{B}ox(\mathcal{U})}{\mathcal{T} <:: \mathcal{IB}ox(\mathcal{U})} \tag{3}$$

$$\frac{\mathcal{B}ox(\mathcal{T}) <:: \mathcal{IB}ox(\mathcal{U})}{\mathcal{IB}ox(\mathcal{T}) <:: \mathcal{IB}ox(\mathcal{U})} \tag{4}$$

$$\frac{\mathcal{T} <:: \mathcal{U}}{\mathcal{R}ef\ \mathcal{T} <: \mathcal{R}ef\ \mathcal{U}} \tag{5}$$

Rules 4 and 5 show the subtype relationship between container types and rule 6 describes the relationship between $\mathcal{R}ef$ types.

The :: Relation – L-types and R-types

In order to have a clear separation between containers and values, we introduce a new operator :: for specifying an l-type of variable/syntax variable. $\texttt{x} :: \mathcal{T}$ is read as "variable `x` is of type $\mathcal{T}$ and is in an *l-context*." We call the already existing : operator the r-type operator. $\texttt{x} : \mathcal{T}$ is read as "expression `x` is of type $\mathcal{T}$ and is in an *r-context*." The operators could also be referred to as the "r-type of" and "l-type of" operators. It should be noted that l-context denotes everything that is *assignable* (indicated as a storable memory) and the r-context, on the other hand, denotes everything that is *expressible* (can be produced by an expression). Therefore, there is no r-value of the type $\mathcal{B}ox(\mathcal{T})$.

$$\frac{\texttt{triv x}}{\texttt{x} : \mathcal{T}riv} \tag{6}$$

$$\frac{\texttt{triv x}}{\texttt{x} :: \mathcal{B}ox(\mathcal{T}riv)} \tag{7}$$

$$\frac{\texttt{int x}}{\texttt{x} : \mathit{Int}} \tag{8}$$

$$\frac{\texttt{int x}}{\texttt{x} :: \mathcal{B}ox(\mathit{Int})} \tag{9}$$

Rules 6 and 7 show the l-value and r-value of the type *triv* while rules 8 and 9 specify similar relationships for the *int* type.

$$\frac{\texttt{x} :: \mathcal{B}ox(\mathcal{T})}{\texttt{immut x} : \mathcal{T}} \tag{10}$$

Rule 10 shows the r-value of the immutable syntax variable.

$$\frac{\texttt{x} :: \mathcal{T}}{\texttt{ref x} : \mathcal{R}ef(\mathcal{T})} \tag{11}$$

$$\frac{\texttt{x} :: \mathcal{T}}{\texttt{ref x} :: \mathcal{B}ox(\mathcal{R}ef(\mathcal{T}))} \tag{12}$$

Rules 11 and 12 describe the behavior of the `ref` type constructor when applied on container types.

$$\frac{\mathtt{x} :: \mathcal{T}}{\mathtt{\&\,x} : \mathcal{R}ef(\mathcal{T})} \tag{13}$$

Rule 13 provides the r-type of the referenced container.

$$\frac{\mathtt{x} : \mathcal{R}ef(\mathcal{B}ox(\mathcal{T}))}{\mathtt{x\ @} : \mathcal{T}} \tag{14}$$

$$\frac{\mathtt{x} : \mathcal{R}ef(\mathcal{IB}ox(\mathcal{T}))}{\mathtt{x\ @} : \mathcal{T}} \tag{15}$$

$$\frac{\mathtt{x} : \mathcal{R}ef(\mathcal{T})}{\mathtt{x\ @} :: \mathcal{T}} \tag{16}$$

Rules 14, 15, and 16 show the behavior of the `@` operator under different conditions.

$$\frac{\mathtt{x} :: \mathcal{T} \qquad \mathtt{y} :: \mathcal{U} \qquad \mathcal{T} <:: \mathcal{U} \qquad \mathtt{x} : \mathcal{V}}{\mathtt{x := y} : \mathcal{V}} \tag{17}$$

Rule 17 shows the behavior of the assignment operator.

Notes

For now, we omit rules for $\mathcal{C}on$ types as they only operate on r-values.

For now, we omit rules for $\mathcal{F}un$ types as they only accept r-values. Any variable and/or primitive type has both r-value and l-value (when it comes to primitive types, only r-value). In all cases, the r-value part of the actual parameter is passed when the function is being called.

Example Relationships

`int i`	$\rightarrow i$	$\rightarrow \mathcal{B}ox(\mathit{Int})$
`immut int ii`	$\rightarrow ii$	$\rightarrow \mathcal{IB}ox(\mathit{Int})$
`ref int ri`	$\rightarrow ri$	$\rightarrow \mathcal{B}ox(\mathcal{R}ef(\mathcal{B}ox(\mathit{Int})))$
`immut ref int iri`	$\rightarrow iri$	$\rightarrow \mathcal{IB}ox(\mathcal{R}ef(\mathcal{B}ox(\mathit{Int})))$
`ref immut int rii`	$\rightarrow rii$	$\rightarrow \mathcal{B}ox(\mathcal{R}ef(\mathcal{IB}ox(\mathit{Int})))$
`immut ref immut int irii`	$\rightarrow irii$	$\rightarrow \mathcal{IB}ox(\mathcal{R}ef(\mathcal{IB}ox(\mathit{Int})))$

Type $\mathcal{IB}ox(\mathcal{IB}ox(\mathit{Int}))$ cannot exist. Nested $\mathcal{B}ox$ types are only possible when there is at least one $\mathcal{R}ef$ type.

$$\mathcal{B}ox(\mathcal{T}riv) <:: \mathcal{B}ox(\mathcal{T}riv)$$

$$\mathcal{B}ox(\mathit{Int}) <:: \mathcal{B}ox(\mathit{Int})$$

$$\mathcal{B}ox(\mathcal{T}riv) <:: \mathit{IB}ox(\mathcal{T}riv)$$

$$\mathcal{B}ox(\mathit{Int}) <:: \mathit{IB}ox(\mathit{Int})$$

$$\mathit{IB}ox(\mathcal{T}riv) <:: \mathit{IB}ox(\mathcal{T}riv)$$

$$\mathit{IB}ox(\mathit{Int}) <:: \mathit{IB}ox(\mathit{Int})$$