The Static Semantics of HARPO/L [DRAFT 4a]

Theodore S Norvell
Electrical and Computer Engineering
Memorial University

December 21, 2010

Abstract

Abstract to be done.

1 Abstract Syntax

We present the abstract syntax of the language as a phrase structured (context-free grammar).

2 Types

2.1 Typing relation

Each well-formed phrase of the language is associated with some phrase type. A context is a mapping from identifiers to phrase types. If E is a phrase of the abstract syntax, t is a phrase type, and Γ is a context, we write

$$\Gamma \vdash E : t$$

to mean that phrase E has type t in context Γ.

For example

$$\Gamma \vdash 1 = 2 : \mathsf{bool}$$

This typing relation is specified by a set of inference rules written

$$\frac{\text{assumptions}}{\text{conclusion}}$$

The domain of the context is always a finite set of identifiers.

The typing relation is intended to define a partial function from contexts and phrases to types

Types and objects

Objects	o	::=	$\text{obj}_{rw}(t)$
Types	t, u, v	::=	$p \mid \text{array}(t) \mid \alpha \,\text{boundedby}\, k \mid k$
Prim. types	p, q	::=	bool | int8 | int16 | int32 | float16 | float32 | float64
Class and interface types	k	::=	$c\,\langle\overline{a}\rangle$
Type variables	α, β		
Generic arguments	a, b	::=	t
Read/Write Mode	rw	::=	r | w
Values	V		
Class identifiers	c, d		

Context

Context	Γ	::=	$x \mapsto o, \Gamma \mid x \mapsto m, \Gamma \mid x \mapsto c, \Gamma \mid x \mapsto (\alpha \,\text{boundedby}\, k)\,, \Gamma \mid \varepsilon$
Methods	m	::=	$[TBD]$

Class environment. A class environment is a partial function from class identifiers to symbol table entries for classes and interfaces. A class or interface symbol table entry records the declarations of the class, the set of interfaces it extends (empty for classes) and the set of interfaces it implements (empty for interfaces).

Class Environment	Θ	::=	$c \mapsto cid, \Theta \mid \varepsilon$
Class and interface declarations	cid	::=	$\lambda\overline{g} \cdot \text{clint}_{ci}(\Delta, \overline{t}, \overline{u})$
Class or interface	ci	::=	class | interface
Generic parameter	g	::=	$x <: t$
Member Declarations	Δ	::=	TBC

3 Building a class environment

We can analyse each class and interface in two passes. The first pass builds a class environment. The second pass does type checking and inference.

In the first pass, we record information about each class and inteface in the class environment (Θ). Since this is done before type checking, all that can be done is to record information in a raw form. For each class declaration

$$\left(\textbf{class}\ x\ \textbf{implements}\ \overline{i}\ D\right)$$

we add an entry

$$c \mapsto \lambda\varepsilon \cdot \text{clint}_{\text{class}}(\Delta, \varepsilon, \overline{u})$$

to the class environment, where c is the fully qualified name for the class, Δ is derived from D, and $\overline{u}$ is derived from $\overline{i}$. Similarly for each interface declaration

$$\left(\textbf{interface}\ x\ \textbf{extends}\ \overline{i}\ D\right)$$

we add an entry

$$c \mapsto \lambda\varepsilon \cdot \mathrm{clint}_{\mathrm{class}}(\Delta, \bar{t}, \varepsilon)$$

to the class environment, where c is the fully qualified name for the interface, Δ is derived from D, and $\bar{t}$ is derived from $\bar{i}$.

When there are generic parameters,

$$\textbf{type } x_i <: E_i$$

we create new type variables α_i and add constraints $\alpha_i <: t_i$ to between the λ and the $\cdot$. Each t_i is derived from each E_i by replacing identifiers representing classes with the corresponding class identifier, replacing braces with angle brackets, replacing each x_i with the coorsponding α_i and so on. [To do: Formalize this.]

Deriving Δ from the sequence of declarations D is done by a similar process. The type expressions used in field declarations, method declarations, and constructor arguments are turned into types t using a superficial analysis. [To do: Formalize this.]

After the first pass is completed for the whole program, we can do full type checking on the whole program.

4 Types of expressions

4.1 Identifiers are looked up in the context

The type of an identifier can be looked up in the context. This is the only rule for identifiers, so an identifier not in the current context results in a type error.

$$\frac{E \text{ is an identifier} \qquad E \in \mathrm{dom}(\Gamma)}{\Gamma \vdash E : \Gamma(E)} \qquad \text{(LOOKUP)}$$

4.2 Constants

For constants of the language we have

$$\frac{E \text{ is an integer constant in } \{-128, ..., +127\}}{\Gamma \vdash E : \mathrm{obj}_{\mathrm{r}}(\mathsf{int8})}$$

$$\frac{E \text{ is an integer constant in } \{-2^{15}, ..., +2^{15}-1\}}{\Gamma \vdash E : \mathrm{obj}_{\mathrm{r}}(\mathsf{int16})}$$

$$\frac{E \text{ is an integer constant in } \{-2^{31}, ..., +2^{31}-1\}}{\Gamma \vdash E : \mathrm{obj}_{\mathrm{r}}(\mathsf{int32})}$$

[TBD: Similar for float]

4.3 Arithmetic expressions

Generally, unary expressions leave the type alone, while binary expressions require the operands to have the same type and produce the same result type. When the operand types are different, there must be a widening conversion from one to the other.

Subtypes are given by the following rules: [[Does this make sense?]]

$$\frac{}{\mathsf{int8} <: \mathsf{int16}} \qquad \frac{}{\mathsf{int16} <: \mathsf{int32}}$$

$$\frac{}{\mathsf{float16} <: \mathsf{float32}} \qquad \frac{}{\mathsf{float32} <: \mathsf{float64}}$$

Furthermore, subtyping is transitive and reflexive

$$\frac{t <: u \qquad u <: v}{t <: v}$$

$$\frac{}{t <: t}$$

All primitive types are subtypes of the built-in interface primitive.

$$\frac{}{p <: \mathsf{primitive}\,\langle\rangle}$$

The following two rules illustrate the typing rules for the binary arithmetic operations on integers. The rules show that either operand may be widened, but not both.

$$\frac{\Gamma \vdash E : \mathrm{obj}(p) \qquad \Gamma \vdash F : \mathrm{obj}(q) \qquad p <: q \qquad p, q \in \{\mathsf{int8}, \mathsf{int16}, \mathsf{int32}\} \qquad \oplus \in \{+, -, *, \mathrm{div}, \mathrm{mod}\}}{\Gamma \vdash E \oplus F : \mathrm{obj}_{\mathrm{r}}(q)}$$

$$\frac{\Gamma \vdash E : \mathrm{obj}(p) \qquad \Gamma \vdash F : \mathrm{obj}(q) \qquad q <: p \qquad p, q \in \{\mathsf{int8}, \mathsf{int16}, \mathsf{int32}\} \qquad \oplus \in \{+, -, *, \mathrm{div}, \mathrm{mod}\}}{\Gamma \vdash E \oplus F : \mathrm{obj}_{\mathrm{r}}(p)}$$

[Arithmetic expressions to be completed.]

4.4 Arrays

Arrays can be indexed by integers

$$\frac{\Gamma \vdash E : \mathrm{obj}_x(\mathrm{array}(t)) \qquad \Gamma \vdash F : \mathrm{obj}(p) \qquad p \in \{\mathsf{int8}, \mathsf{int16}, \mathsf{int32}\}}{\Gamma \vdash E[F] : \mathrm{obj}_x(t)}$$

4.5 Inheritence

Classes can implement interfaces, while interfaces can extend other interfaces. In the future we may allow classes to extend classes, so these rules are written with that in mind.

Extension and implementation induce a subtype relation on classes and interfaces as follows

- Inheritence by extension

$$\frac{\Theta(c) = \lambda \overline{g} \cdot \mathrm{clint}_{ci}(\Delta, \overline{t}, \overline{u}) \quad \exists t \in \overline{t} \cdot d\left\langle \overline{b} \right\rangle = t[\overline{g} := \overline{a}]}{c\left\langle \overline{a} \right\rangle <: d\left\langle \overline{b} \right\rangle}$$

- Inheritence by implementation

$$\frac{\Theta(c) = \lambda \overline{g} \cdot \mathrm{clint}_{ci}(\Delta, \overline{t}, \overline{u}) \quad \exists u \in \overline{u} \cdot d\left\langle \overline{b} \right\rangle \in u[\overline{g} := \overline{a}]}{c\left\langle \overline{a} \right\rangle <: d\left\langle \overline{b} \right\rangle}$$

Furthermore, a type variable is a subtype of its bound

$$\frac{}{(\alpha \,\mathrm{boundedby}\, k) <: k}$$

As noted earlier, subtyping is reflexive and transitive.

4.6 Fields and methods

A field can be found in an object that implements an interface or class that declares the field. The same rule serves for method lookup. Fields and methods may also be inherited. Rules on consistancy of inheritence (see section [[TBD]]) ensure that a field or method can only be inherited from one supertype and that there is no conflict between the declarations of a type and any of its supertypes.

$$\frac{\Gamma \vdash E : \mathrm{obj}_{rw}(t) \quad t <: x\left\langle \overline{a} \right\rangle \quad \Theta(x) = \lambda \overline{g} \cdot \mathrm{clint}_{ci}(\Delta, \overline{u}, \overline{v}) \quad \Delta(i) = (\mathrm{public}, om)}{\Gamma \vdash E.i : om[\overline{g} := \overline{a}]}$$

4.7 Initialization Expressions

A new object can be created from a concrete class

$$\frac{\Gamma \vdash E : c\left\langle \overline{a} \right\rangle \quad [\text{Matching constructor arguments is To Be Done.}]}{\Gamma \vdash \mathbf{new}\ E(F_0, F_1, ..., F_{n-1}) : \mathrm{obj}_{\mathrm{w}}(c\left\langle \overline{a} \right\rangle)}$$

A new array can be created using a for loop.

$$\frac{\Gamma \vdash E : \mathrm{obj}(t) \qquad t <: \mathsf{int}\left\langle\right\rangle \qquad \Gamma_{i \leftarrow \mathrm{obj}_{\mathrm{r}}(q)} \vdash F : \mathrm{obj}(t)}{\Gamma \vdash (\mathbf{for}\ i : E\ \mathbf{do}\ F) : \mathrm{obj}_{\mathrm{w}}(\mathrm{array}(t))}$$

It is required that E be a compile time constant, evaluable after generic specialization. This requirement is not captured formally by this rule.

A choice of initializations is given by an 'if' expression

$$\frac{\Gamma \vdash E : \mathrm{obj}(\mathsf{bool}) \qquad \Gamma \vdash F : \mathrm{obj}(t) \qquad \Gamma \vdash G : \mathrm{obj}(t)}{\Gamma \vdash (\textbf{if } E \textbf{ then } F \textbf{ else } G) : \mathrm{obj}_{\mathrm{w}}(t)}$$

Other initializations are simply expressions and are typed the same as other expressions.

5 Type checking types

Some of the phrases in a program represent types.

5.1 Primitives

Each primitive type is typed to itself

$$\frac{p \in \{\mathsf{bool}, \mathsf{int8}, \mathsf{int16}, \mathsf{int32}, \mathsf{float16}, \mathsf{float32}, \mathsf{float64}\}}{\Gamma \vdash p : p}$$

5.2 Class and interfaces

In the abstract syntax, class names are followed by 0 or more generic arguments in braces. (In the concrete syntax, the baces are omitted in the 0 argument case.)

Calculating the type of a phrase $x\,\{E_0, E_1, ...E_{n-1}\}$ is done in several steps

- Look up identifier x in the context. It should map to a class identifier, c.
- Look up that class identifier in the class environment. This gives a lambda expression, which should have n generic parameters.
- Calculate the type of each phrase E_i giving a type a_i.
- Check that each argument type a_i matches the coresponding generic argument.
- The resulting class type is $c\,\langle a_0, a_1, ..., a_{n-1}\rangle$.

$$\frac{\begin{array}{c}\Gamma(x) = c \\ \Theta(c) = \lambda\overline{\alpha} <: \overline{t} \cdot \mathrm{clint}_{ci}(\Delta, \overline{u}, \overline{v}) \\ \Gamma \vdash E_i : a_i, \text{ for all } i \\ a_i <: t_i[\overline{\alpha} := \overline{a}], \text{ for all } i\end{array}}{\Gamma \vdash x\left\{\overline{E}\right\} : c\,\langle\overline{a}\rangle}$$

5.3 Array types

Phrases representing array types include a bound. This bound must be a compile time constant calculable after generic expansion. Our rule here does not capture that requirement, as it can only be determined at or after specialization

$$\frac{\Gamma \vdash E : t \qquad \Gamma \vdash F : \mathrm{obj}(u) \qquad u <: \mathsf{int}\,\langle\rangle}{\Gamma \vdash E[F] : \mathrm{array}(t)}$$

5.4 Generic parameters

Inside a generic class or interface the parameters' identifiers will be bound —in the context— to generic parameters of the form

$$\alpha \,\mathrm{boundedby}\, k$$

6 Type checking of commands

For statements, I'll use judgements of the form

$$\Gamma \vdash E$$

where E is a command, to mean that E is well typed. We can think of this as an abbreviation for $\Gamma \vdash E : \mathsf{comm}$, where comm is the type of commands.

6.1 Assignments

Assignments are permitted only for primitive variables. Thus the rule is

$$\frac{\Gamma \vdash E : \mathrm{obj}_{\mathrm{w}}(t) \qquad \Gamma \vdash F : \mathrm{obj}(u) \qquad u <: t \qquad t <: \mathsf{primitive}\,\langle\rangle}{\Gamma \vdash E := F}$$

6.2 Local variable declaration

Local variables may be of any object type

$$\frac{\Gamma \vdash E : t \qquad \Gamma \vdash F : \mathrm{obj}(u) \qquad u <: t \qquad \Gamma_{i \leftarrow \mathrm{obj}_{\mathrm{w}}(t)} \vdash S}{\Gamma \vdash \mathbf{obj}\; i : E := F\; S}$$

For local variables, the type, if omitted, is inferred from the type of the expression.

$$\frac{\Gamma \vdash F : \mathrm{obj}(t) \qquad \Gamma_{i \leftarrow \mathrm{obj}_{\mathrm{w}}(t)} \vdash S}{\Gamma \vdash \mathbf{obj}\; i := F\; S}$$

6.3 Blocks

A block is a sequence of 0 or more statements.

$$\frac{\Gamma \vdash S_i, \text{ for all } i \in \{0, 1, ..., n-1\}}{\Gamma \vdash S_0\, S_1 \ldots S_{n-1}}$$

6.4 Method calls

TBD

6.5 Sequential control flow

$$\frac{\Gamma \vdash E : \mathrm{obj}(\mathsf{bool}) \qquad \Gamma \vdash S \qquad \Gamma \vdash T}{\Gamma \vdash (\textbf{if } E \textbf{ then } S \textbf{ else } T)}$$

$$\frac{\Gamma \vdash E : \mathrm{obj}(\mathsf{bool}) \qquad \Gamma \vdash S}{\Gamma \vdash (\textbf{wh } E \textbf{ do } S)}$$

$$\frac{\Gamma \vdash E : \mathrm{obj}(t) \qquad t <: \mathsf{int}\,\langle\rangle \qquad \Gamma_{i \leftarrow \mathrm{obj_r}(t)} \vdash S}{\Gamma \vdash (\textbf{for } i : E \textbf{ do } S)}$$

6.6 Parallelism

$$\frac{\Gamma \vdash S_i, \text{ for all } i \in \{0, 1, ..., n-1\}}{\Gamma \vdash (\textbf{co } S_0 \mid\mid S_1 \mid\mid ... \mid\mid S_{n-1})}$$

$$\frac{\Gamma \vdash E : \mathrm{obj}(t) \qquad t <: \mathsf{int}\,\langle\rangle \qquad \Gamma_{i \leftarrow \mathrm{obj_r}(t)} \vdash S}{\Gamma \vdash (\textbf{co } i : E \textbf{ do } S)}$$

6.7 Method Implementation

TBD

6.8 Atomicity

$$\frac{\Gamma \vdash S}{\Gamma \vdash (\textbf{atomic } S)}$$

7 Type Checking Declarations

7.1 Class declarations

7.2 Interface declarations

7.3 Global object and field declarations

7.4 Method declarations