P3 Field Descriptor Definitions

Peter Vanderbilt
pvanderbilt@acm.org

Version 3.1.4

Abstract

This technical note describes the syntax and meaning of P3 field descriptor definitions. These appear in the body of a class or module definition to describe the fields of instances. These fields can contain complex content, such as types and hybrids.

This technical note is one of a collection that describes the research of the author; see [4] for more details.¹

1 Introduction

The bodies of P3 class and module definitions consist of a sequence of field descriptor definitions (see P3 Class Definitions⇑ [2]). To simplify the exposition, let us use the term template to refer to either a class or module. Actually, a class can have two templates, one for member fields and one for static fields. In general, a template definition names a “super-template” (a superclass or super-module) and has zero or more field descriptor definitions. A template’s field descriptor definitions combine with the field descriptors of the super-template to yield a set of field descriptors.

A field descriptor definition is a P3 construct used to create a field descriptor. A field descriptor provides some or all of the following information about a field:

- its name,
- an expression for its current value (an implementation),
- the type of its current value, and
- a type that constrains its future values.

The current value of a template’s field is the one appearing in an instance of the template. A future value of a field is one provided in some (possibly indirect) extension of the containing template and a future type is a type constraining future values. For data fields, the current and future types are the same, but they differ for complex content.

There are two kinds, or modes, of field descriptors:

¹Copyright © 2014-2019 by Peter Vanderbilt. ©οοοο This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit creativecommons.org/licenses/by-nc-nd/4.0/.
• A *virtual* field descriptor has all the information mentioned above: a name, a typed implementation and a future type.

• An *abstract* field descriptor has just a name and a future type (but no implementation).

We use the following terms:

• A field descriptor of a given name is *pre-existing* if it exists in the super-template (either because it was defined there or in a higher template).

• A field descriptor definition creates a *new* field iff there is no pre-existing field descriptor of that name in the super-template.

• A field descriptor definition modifies or replaces a pre-existing field descriptor when the names match. If the field descriptor definition includes an implementation, that implementation either *overrides* or *implements* the one of the super field descriptor. If the field descriptor definition includes a future type, it *refines* the one of the super field descriptor (by replacing it with a subtype).

• If a field descriptor is pre-existing and there is no field descriptor with the same name in the template definition, the one from the super-template *inherits* into the template.

2 Syntax of field descriptor definitions

2.1 Grammar of field descriptor definitions

The following grammar defines non-terminal $J$ to range over field descriptor definitions:

$$J ::= \text{new virtual } V \mid \text{new abstract } L \mid (\text{override } | \text{impl}) \text{ virtual } J \mid \text{refine } (\text{override } | \text{impl}) \text{ virtual } V \mid \text{refine abstract } L.$$

See *P3 Preliminaries* [3] for a description of P3’s grammar and conventions. Non-terminal $V$ will be described in subsection 2.2. Non-terminal $L$ ranges over declarations and $J$ over definitions, which are basic P3 concepts – however the defining technical note is in progress. The qualifier *new* requires that the name being defined must not pre-exist in the super-template; if one of the other qualifiers is present instead, the name must pre-exist. In the convenient form, the *new* and *virtual* keywords may be omitted.

**Result:** A *virtual* field descriptor definition results in a virtual field descriptor (as described in section 1) and an *abstract* field descriptor definition results in an abstract one.

• The qualifier *new* means that this definition creates a new field descriptor.
• The qualifier **override** means that the super-template has a virtual field descriptor (with the given name) and this definition replaces its implementation.

• The qualifier **impl** means that the super-template has an abstract field descriptor (with the given name) and this definition provides an implementation.

• The qualifier **refine** means that this definition replaces the future type of the super-template’s field descriptor (with a subtype).

If the resulting template has any abstract field descriptors, either newly defined, inherited or refined, it is a *partial* template. If the resulting template has only virtual field descriptors, it is *concrete*.

### 2.2 Virtual field specifications

A virtual field specification describes the content of a virtual field descriptor. In general, it needs to specify both the current and future aspects of a field. The following grammar defines non-terminal $\mathcal{V}$ (which appears in (2.1)) to range over virtual field specifications:

$$
\mathcal{V} ::= J^D \mid \mathcal{V}^M \mid \mathcal{V}^H.
$$

The non-terminal $J^D$ ranges over P3 type-announcing data definitions and $\mathcal{V}^M$ and $\mathcal{V}^H$ will be defined by (2.3) and (2.4), respectively.

A type-announcing data definition is any definition that can be rewritten to “\texttt{const x : T = e}”. When used in a virtual field definition, it yields a virtual field descriptor with name $x$, implementation $e$ and both current and future types being $T$.

The non-terminal $\mathcal{V}^M$ ranges over meta-valued virtual field descriptors. Its grammar is as follows:

$$
\mathcal{V}^M ::= \text{meta } X :: K : Z = M | \ldots . \tag{2.3}
$$

The **meta** form defines a virtual type-level field whose name is $X$, whose kind is $K$, whose value is $M$ and whose future values are constrained to satisfy meta-type $Z$.

The ellipsis in (2.3) denotes convenient form alternatives which are not currently written up, but they include (a) forms that start with **type** and (b) forms with parameters to the left of the equals (which denote meta-functions). For example, the following are two convenient-form descriptor definitions and their canonical-form equivalents:

- \texttt{type X : Z = T \approx meta X :: * : Z = T}
- \texttt{type +X = T \approx meta X :: * : \{?<:T\} = T}

The first form defines a virtual type field named $X$ whose value is $T$ and whose future values are constrained to satisfy the type $Z$. The second form defines a virtual type field named $X$ whose current value is $T$ and whose future values must subtype $T$. Kind “*” is the kind of types and $\{?<:T\}$ is a type whose instances are the subtypes of $T$. Other forms of virtual field descriptor definitions can be added to the language by specifying their canonical form equivalents.

Non-terminal $\mathcal{V}^H$ ranges over hybrid virtual field definitions:
The `hconst` form defines a virtual hybrid constant whose name is `x`, whose kind is `K`, whose value is `e` which must be of (hybrid) type `T_c` and whose future values must satisfy the type `T_f`.

Notice that in the `hconst` form, the current type, `T_c`, is explicitly required, in addition to the future type, `T_f`. This is because, to simplify the logic, definitions in a recursion must be explicitly typed. Actually, the syntax for virtual hybrids is tentative; it would be nice to have a cleaner form.

### 3 Translation to current/future form

We take the approach that each field definition can be expressed as a current definition or a future declaration or both. A current definition of a field defines its implementation and has the following form:

```
current x :: K_c : T_c = e
```

Non-terminal `x` ranges over identifiers, `K_c` over kinds, `T_c` over types and `e` over expressions. This defines a field whose name is `x` and whose value is that resulting from `e`. `K_c` and `T_c` are kind and type announcements, which are intended properties of field `x`.

A future declaration specifies what must be true of a field for this class and for the class’s extensions. It has the following form:

```
future x :: K_f : T_f
```

Again `x` ranges over identifiers, `K_f` over kinds and `T_f` over types. This specifies that the field `x` must be of kind `K_f` and type `T_f` in all extensions of this class (including the current class if not abstract).

The following subsections describe how field specification definitions are mapped to current and future forms.

### 3.1 Rewrite virtual fields

Each virtual field descriptor definition has both current and future aspects. This section describes how virtual descriptors can be rewritten to current/future form. A rule is given for each level of content: data, meta and hybrid. The rule shows how a canonical definition is rewritten; a convenient definition is rewritten to canonical form and then the rule is applied. The rules are written using “new virtual” but they also apply to the following:

```
refine override virtual
refine impl virtual
```

For non-complex (data) virtual fields, the current and future types are the same. The following equivalence shows how a canonical virtual data definition is represented in its two aspects:
new virtual const \( x : T = e \approx \) \( (3.1) \)
current \( x :: e : T = e \);
future \( x :: e : T \)

The symbol \( e \) denotes the empty kind, the kind of pure data. Note that the current and future types are the same.

The following is the rewrite rule for the canonical form virtual meta-definition:

new virtual meta \( X :: K : Z = M \approx \) \( (3.2) \)
current \( X :: K : \{\{?\rightarrow M\}\} = M \);
future \( X :: K : Z \)

Note that the current type is one that captures the current value of \( M \) because \( \{\{?\rightarrow M\}\} \) is the meta-type whose instances are the meta-expressions that equal \( M \).

The rewrite for a virtual hybrid constant definition is as follows:

new virtual hconst \( x :: K : T_f = (T_c) e \approx \) \( (3.3) \)
current \( x :: K : T_c = e \);
future \( x :: K : T_f \)

By design, a virtual hconst has both future and current types included, exactly so that it can be decomposed as above.

### 3.2 Rewrite abstract fields

Abstract fields declare a future field without giving its implementation. Thus, fields marked abstract rewrite to just future definitions (with no current part):

abstract const \( x :: T \approx future x :: e : T \) \( (3.4) \)
abstract meta \( X :: K : Z \approx future X :: K : Z \) \( (3.5) \)
abstract hconst \( x :: K : T_f \approx future x :: K : T_f \) \( (3.6) \)

These rewrites apply to abstract fields qualified by either new or refine.

### 3.3 Rewrite implementation fields

Virtual fields descriptors marked override or impl provide implementations. When not marked refine, they have no future aspect. Thus they rewrite to just current definitions.

The canonical form of a data implementation field is as follows, with its equivalent:

(override | impl) virtual const \( x = e \approx \) \( (3.7) \)
current \( x :: e : T = e \)

where \( T \) is the type of field \( x \) in the super-template. (The field must pre-exist since it isn’t marked new. Either current or future type will do since they are equal.)

The canonical form of a meta-constant implementation field is as follows, with its equivalent:
The canonical form of a hybrid constant implementation field is as follows, with its equivalent:

\[
\text{(override | impl) virtual hconst } x :: K : T_c = e \approx (3.9)
\]

\[
\text{current } x :: K : T_c = e
\]

**History by version number**

**3.1.4 (Apr 2014)** Fixed PDF URLs (for new website).

**3.1.3 (Feb 2014)** Added copyright and Creative Commons license.

**3.1.2 (Jul 2013)** Added reference to *P3 Preliminaries* \[3\]. Also clarified text and fixed some jumps.

**3.1.1 (Jul 2013)** Fixes to cross-references.

**3.1 (Jul 2013)** Initial version formed from text from the class definitions and complex encoding technotes. Text was moved or replaced as needed and clarifying text was added. Other text was rewritten.

**References**


