Theme-D Language Manual

Tommi Höynälänmaa

November 8, 2017

Contents

List of Figures

Chapter 1

Introduction

The purpose of programming language Theme-D is to extend Scheme with static typing. Theme-D has an object system with single inheritance and multi-methods. Theme-D also has parametrized types and parametrized procedures. *Translation* shall mean the compilation and linking of a Theme-D program. Theme-D is mainly intended to be a compiled language. The standard of programming language Scheme can be found at [3]. Theory of type systems in functional programming languages can be found at [2]. Homepage for guile can be found at http://www.gnu.org/software/guile/. Homepage for Scheme48 is located in http://s48.org/. Homepage for the functional programming language ocaml is located in http://caml.inria.fr/. Theme-D resembles Jaap Weel's Theme [4] but Theme-D is more dynamic and the objects in Theme-D need to have type tags. I remember seeing a programming language called "bits", extending Scheme by a static type system, but I was unable to find it again.

Chapter 2

Hello World

Here is the implementation of “Hello World” in Theme-D:

```
(define-proper-program hello-world

   (import (standard-library core)
           (standard-library console-io))

   (define main
         (lambda (() <none> nonpure)
               (console-display-line "Hello, world!"))))
```

Chapter 3

Programs and Modules

All the code in Theme-D is organized into *units*. A unit is either a *program*, an *interface* or a *body*. A program is either a *proper program* or a *script*. A combination of an interface and the body that implements it is called a *module*. See sections 6.8.1, 6.8.2, 6.8.3, and 6.8.4 for the syntax for defining units.

A proper program has to define a procedure called called `main`. The accepted argument types and result type of `main` depend on the target platform. But every Theme-D implementation is required to accept the following for `main`:

1. Result type `<none>` or `<integer>`

2. Empty argument list, argument list consisting of one argument with type `(:uniform-list <string>)`, or a single argument of a tuple type consisting only of `<strings>`.

When a proper program is executed all the toplevel expressions in the program and in the modules it imports are executed and then the procedure `main` is called. A script contains no `main` procedure. When a script is executed all the toplevel expressions in the program and in the modules it imports are executed.

An interface contains all the definitions or declarations for the variables that the module exports. An interface contains only declarations for the procedures and the parametrized procedures that the module exports. A body contains definitions of all private variables of the module and definitions of all the procedures and the parametrized procedures declared in the interface. Both the interface and the body may import other modules using keyword **import** or **import-and-reexport**. An interface may reexport variables imported from other modules.

The module imports between the interfaces may not be cycled. I.e. if an interface A imports module B directly or indirectly the interface of B may not import module A. However, the body of B may import module A.

When an interface of a module A imports other modules the definitions and declarations in the imported modules do not become visible automatically when the module A is imported. However, an interface may contain **reexport** statements, which export a variable imported from another interface. An interface may also contain **import-and-reexport** statements, which import a module and reexport all the variables it exports. The variables imported into an interface become visible in the corresponding body automatically. A body always imports the interface of the module implicitly. This import may not be specified explicitly in the **import** clause of the body. Modules can also be used without importing its contents into the toplevel namespace. This is done with keyword **use**. The variables in this kind of modules are accessed with syntax `(@` *module variable*`)`.

An interface must not contain any toplevel procedure calls. A body or a program may contain toplevel procedure calls. A body or a program containing toplevel procedure calls must ensure that the called procedures are linked properly using the form **prelink-body**, see section 6.8.11.

Chapter 4

Variables, Objects, and Types

4.1 Variables

A variable whose value cannot be changed is called a *constant*. A variable whose value can be changed is called a *mutable variable*. A *volatile variable* is a mutable variable that can be changed by pure expressions (expressions without side effects). Note that it is possible to change the components of a constant, e.g. setting elements of a constant vector. Variables are lexically scoped as in Scheme.

4.2 Classes and Logical Types

Every Theme-D object has a *static type* and a *dynamic type*. The dynamic type of a Theme-D object is always a *class*. Types that are not classes are called *logical types*.

A type may *inherit* from another type. A type always inherits from itself. When type A inherits from type B and variable `y` has been declared with type B a value `y` of type A can be assigned to `y`. We write $A :< B$ to mean that A inherits from B. When the static or dynamic type of a value or a variable `y` is A and A inherits from B we say that `y` is *instance* of type B. Every type except `<none>` inherits from the class `<object>`. Every class is an instance of class `<class>`. Class `<class>` is an instance of itself. A class whose instances are classes is called a *metaclass*.

Every class that is not an instance of a parametrized class is called a *simple class*. Every simple class except `<object>` and `<none>` has an *immediate superclass*, which is itself a class. We write $A ::< B$ to mean that B is the immediate superclass of A. A class A inherits from a class B if and only if $A ::< B$ or there exists a finite sequence X_1, ..., X_n consisting of classes so that $A ::< X_1 ::< ... ::< X_n ::< B$. The dynamic type of an object y is always a subtype of the static type of y. See section 6.9.2 for the syntax for defining new classes. See subsection 4.13 for the algorithm that checks if one type is a subtype of another type.

Every class has the following boolean-valued attributes:

- *inheritable*
- *immutable*
- *equality by value*

A class is inheritable if and only if it is allowed to be a superclass of another class. If a class is immutable no fields of instances of the class can be changed. If a class is equal by value two instances of the class are equal if and only if all of their fields are equal. Otherwise instances of a class are equal if and only if they are the same object.

Each field of a class has a name, type, read access specifier, write access specifier, and an optional initial value. Possible values of the access specifiers are `public`, `module`, and `hidden`. Specifier `public` means that the field is accessible everywhere. Specifier `module` means that the field is accessible only in the same module where the class is defined. Specifier `hidden` means that the field is accessible nowhere. Its value can be set in object creation (in **make** expression), though. A field may have an initial value, which has to be an instance of the type of the field. When an object is created with **make** only the values of fields without an initial value are given as the arguments of **make**. Keyword **make** actually calls the constructor of a class in order to create an object. Expression `(make` *class* arg_1 ...arg_n `)` is equivalent to `((`**constructor** *class* `)` arg_1 ...arg_n `)`.

The access of a constructor is specified in a similar way. If a constructor is not visible somewhere keyword **make** cannot be used for the class at that position. Note that if you want to define an abstract class which can be inherited but not instantiated define the constructor access to `hidden`.

A class may define a zero value, which can be accessed with syntax `(zero` *class*`)`, see section 6.16.3. This is useful for parametrized numerical classes. A parametrized class may define a zero value for its instances, see file `theme-code/tests/test220.thp`. For example, vector addition can be implemented as follows:

```
(define-param-proc my-sum (%number)
        (((v1 (:mutable-value-vector %number))
        (v2 (:mutable-value-vector %number)))
        (:mutable-value-vector %number)
        (force-pure))
   (let ((len1 (mutable-value-vector-length v1))
         (len2 (mutable-value-vector-length v2)))
      (assert (= len1 len2))
      (let ((result (make-mutable-value-vector
                     %number len1 (zero %number))))
        (do ((i <integer> 0 (+ i 1)))
            ((>= i len1))
          (mutable-value-vector-set!
            result i
            (+ (mutable-value-vector-ref v1 i)
```

```
          (mutable-value-vector-ref v2 i))))
result)))
```

A diagram about an example simple class inheritance hierarchy is presented in figure 4.1. A thick line means “A is an instance of B” and a thin line “A inherits from B”. A rectangle means a class and a circle a non-class object.

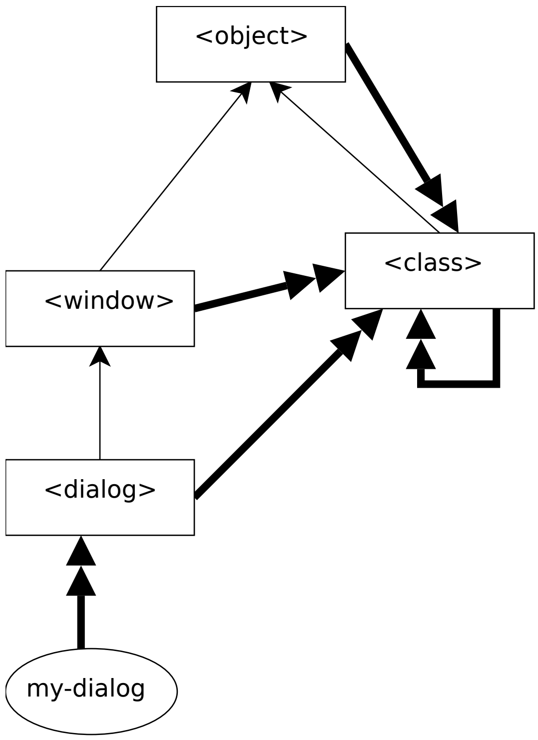

Figure 4.1: Example inheritance hierarchy for simple classes.

Logical types are specified simply be defining a constant whose value is some type. Here is an example of a logical type definition:

```
(define <my-type> (:union <real> <integer>))
```

4.3 Parametrized Types

See sections 6.9.8 and 6.9.7 for the syntax of parametrized type definitions. Parametrized types are types that have *type parameters*. When values (types) are assigned to the type variables we get an instance of the parametrized type. Instances of parametrized classes are classes and instances of parametrized logical types are logical types. Instances are created with syntax

(*parametrized-type type-parameter*$_1$ ...*type-parameter*$_n$)

A diagram about an example parametrized class inheritance hierarchy is presented in figure 4.2. A thick line means "A is an instance of B" and a thin line "A inherits from B". A rectangle means a class and a circle a non-class object.

4.4 Signatures and Parametrized Signatures

A signature is a data type defined by specifying the procedures that the object belonging to the signature has to implement. They resemble Java interfaces but signatures are multiply dispatched. Parametrized signatures are singatures parametrized by type parameters. See sections 6.9.10 and 6.9.12.

If an application of a procedure contains signatures as an argument type we use the following algorithm to check if the application is valid:

1. Substitute keyword **this** by the signature itself in all the procedure specifiers referring to the same procedure as the procedure to be called.

2. Check that the application argument type list is a subtype of an argument type of some of the substituted procedure specifiers.

See section 8.1 for examples about signatures.

4.5 Built-in Classes

The classes listed in this section are also called *primitive classes*. An instance of a primitive class is called a *primitive object*.

4.5.1 `<object>`

Every value in Theme-D is an instance of `<object>`. Every type except `<none>` is a subtype of `<object>`. Class `<object>` defines no fields. Class `<object>` is inheritable, immutable, and not equal by value. Note that subclasses of `<object>` do not need to be immutable.

4.5.2 `<class>`

Every class in Theme-D is an instance of `<class>`. Class `<class>` is an instance of itself. Class `<class>` is inheritable, immutable, and not equal by value.

4.5.3 `<integer>`

Instances of class `<integer>` are integer numbers. Class `<integer>` is immutable, equal by value, and not inheritable.

4.5.4 `<real>`

Instances of class `<real>` are real numbers. Class `<real>` is immutable, equal by value, and not inheritable. Note that `<integer>` objects are not instances of `<real>`.

4.5.5 `<boolean>`

Boolean values are similar to Scheme boolean values. Class `<boolean>` is immutable, equal by value, and not inheritable.

4.5.6 `<null>`

Class `<null>` is the class of an empty list. The empty list object is denoted by `null` or `()` and it behaves similarly to the empty list in Scheme. Class `<null>` is immutable, equal by value, and not inheritable. Note that if you use notation `()` you usually have to quote it as in Scheme.

4.5.7 `<symbol>`

Symbols are similar to Scheme symbols. Class `<symbol>` is immutable, equal by value, and not inheritable.

4.5.8 `<string>`

Strings are similar to Scheme strings. Class `<string>` is immutable, equal by value, and not inheritable.

4.5.9 `<character>`

Characters are similar to Scheme characters. Class `<character>` is immutable, equal by value, and not inheritable.

4.5.10 `<eof>`

Class `<eof>` is the class of end-of-file object, which is similar to the Scheme end-of-file object. There are no other instances of `<eof>`. Class `<eof>` is immutable, equal by value, and not inheritable. The end-of-file object is denoted by `eof`.

4.6 Built-in Logical Types

4.6.1 `<type>`

Every type (class or logical type) in Theme-D is an instance of `<type>`.

4.6.2 `<none>`

No object in Theme-D is an instance of `<none>`. The result type of a procedure returning no value shall be `<none>`.

4.7 Built-in Parametrized Classes

The builtin parametrized classes are:

- `:procedure`
- `:simple-proc`
- `:param-proc`
- `:gen-proc`
- `:vector`
- `:mutable-vector`
- `:value-vector`
- `:mutable-value-vector`
- `:pair`

See chapter 5 for descriptions of the procedure classes. See subsection 4.10 for descriptions of the vector classes. See subsection 4.11 for descriptions of pairs.

4.8 Built-in Parametrized Logical Types

4.8.1 `:union`

Let u be a union type created by (`:union` a_1 ...a_n). Let t_1 ...t_m be the translated argument list generated from a_1 ...a_n , see section 5.7. An object *obj* is an instance of u if and only if *obj* is an instance of some t_k , k = 1, ..., n . Object *obj* is allowed to be an instance of multiple component types $t_{k'}$.

4.8.2 `:uniform-list`

Let u be a uniform list type created by (`:uniform-list` a). Let (t) be the translated argument list generated from (a). Objects of logical type u are lists having elements of type t. A parametrized logical type equivalent to `:uniform-list` can be created with code

```
(declare :my-list <param-logical-type>)
(define-param-logical-type :my-list (%type)
  (:union (:pair %type (:my-list %type)) <null>)
```

4.9 Recursive Definitions

In general, when you define a variable recursively you have to forward declare it. However, forward declaration is not needed with **define-procedure** and **define-param-proc**. Notice how a forward declaration of a logical type is done in the following case:

```
(declare <my-list> :union)
(define <my-list> (:union (:pair <integer> <my-list>) <null>))
```

4.10 Vectors

4.10.1 General

Parametrized classes `:vector`, `:mutable-vector`, `:value-vector`, and `:mutable-value-vector` are called *vector metaclasses*. Instances of vector metaclasses are called *general vector classes*. Objects of general vector classes are called *general vectors*.

4.10.2 Normal Vectors

Instances of `:vector` are called *normal vector classes*. Objects of class (`:vector` *t*) are immutable one-dimensional vectors having elements of type *t*. See subsections 7.4.16 and 7.4.12 for the creation of vectors. The first element of a vector has index 0.

4.10.3 Mutable Vectors

Instances of `:mutable-vector` are called *mutable vector classes*. Objects of class (`:mutable-vector` *t*) are mutable one-dimensional vectors having elements of type *t*. See subsections 7.4.14 and 7.4.10 for the creation of mutable vectors. The first element of a mutable vector has index 0.

4.10.4 Value Vectors

Instances of `:value-vector` are called *value vector classes*. Class `:value-vector` is similar to `:vector` except the instances of `:value-vector` are equal by value. See subsections 7.4.15 and 7.4.11 for the creation of value vectors. The first element of a value vector has index 0.

4.10.5 Mutable Value Vectors

Instances of `:mutable-value-vector` are called *mutable value vector classes*. Class `:mutable-value-vector` is similar to `:mutable-vector` except the instances of `:mutable-value-vector` are equal by value. See subsections 7.4.13 and 7.4.9 for the creation of mutable value vectors. The first element of a mutable value vector has index 0.

4.11 Pairs and Tuples

When a_1 and a_2 are objects the class of the pair (a_1 . a_2) is (`:pair` t_1 t_2) where t_1 is the class of a_1 and t_2 is the class of a_2.

Let u be a pair class created by (`:pair` a_1 a_2). Let (t_1 t_2) be the translated argument list generated from (a_1 a_2). Object of type u is an immutable pair whose first component is of type t_1 and second component of type t_2 . Let a_1 , a_2 , b_1 , and b_2 be type formulas. Let (t_1 t_2) be the translated argument list generated from (a_1 a_2) and (u_1 u_2) the translated argument list generated from (b_1 b_2) Now type (`:pair` a_1 a_2) is a subtype of (`:pair` b_1 b_2) if and only if t_1 is a subtype of u_1 and t_2 is a subtype of u_2 .

A tuple type is a type of a finite sequence of possibly nonuniform objects. Formally, if t_k, $k = 1, \ldots, n$, are types the tuple type (`:tuple` t_1, ..., t_n) is equivalent to (`:pair` t_1 (`:pair` t_2 ... (`:pair` t_n `<null>`) ...)).

4.12 Foreign Function Interface

The semantics of **prim-proc**, **unchecked-prim-proc**, **param-prim-proc**, **unchecked-param-prim-proc**, **define-prim-class**, **define-goops-class**, and **define-normal-goops-class** depend on the Theme-D translation target platform. See 6.14.5 and 6.14.6 for the definition of primitive procedures. The keyword **define-normal-goops-class** is discussed in the standard library reference. If you want to use your own Scheme procedures with these keywords you can specify the Scheme files to be loaded into the runtime Theme-D environment with environment variable `THEME_D_CUSTOM_CODE`. Separate the file names with :'s. See files `theme-d-code/tests/test223.thp` and `runtime/run2.scm` for an example. A Scheme implementation of a parametrized primitive procedure has to take the type parameters as arguments before the proper procedure arguments. See `theme-d-code/tests/test142.thp` and `theme-d-code/tests/aux-my-map.scm` for an example. Custom primitive classes may be defined with keyword **define-prim-class**. See section 6.9.11 and tests `test223`, `test224`, and `test226`. GOOPS classes may be imported into Theme-D with keyword **define-goops-class**. See section 6.9.4 and tests `test279` and `test280`.

Foreign function interface may cause problems with the linker output stripping. For example, suppose you define class `<gtk-widget>` and its subclass `<gtk-window>`. Suppose also that you define procedure `gtk-window-new` in your foreign code so that the procedure returns a `gtk-window` but its declared return type is `<gtk-widget>`. Then it is possible that the linker strips the class `<gtk-window>` off from the target code even though it is needed by the type system. This problem may be solved by a `prevent-stripping` expression. For Theme-D-Gnome this can also be solved by using creator procedures instead of GTK constructors.

The custom primitive classes have to be disjoint with each other and with built-in primitive classes. That is, no object shall belong to two different primitive classes. GOOPS classes may overlap with each other but no two GOOPS classes shall be identical.

4.13 Algorithm to Compute Subtype Relation

4.13.1 IsSubtype

Arguments:

t_1: a type
t_2: another type
M: the set (list) of types already visited

Result:

is-subtype? : #t if t_1 is a subtype of t_2, #f otherwise

Algorithm: IsSubtype[t_1, t_2, M]

1. If t_1 is incomplete or t_2 is incomplete return #t iff t_1 and t_2 are the same object and #f otherwise.
2. If $(t_1, t_2) \in M$ return #t.
3. Set $M' := M \cup \{(t_1, t_2)\}$.
4. If $t_1 = t_2$ return #t.
5. If $t_2 =$ <object> return #t.
6. If t_1 and t_2 are type variables return #t iff t_1 and t_2 are equal.
7. If t_1 and t_2 are primitive classes then return #t iff they are the same object.
8. If t_1 is not a signature and t_2 is a signature return IsSubtypeXSignature[t_1, t_2, M'].
9. If t_1 and t_2 is are signatures return IsSignatureSubtype[t_1, t_2, M'].
10. If t_1 is a union return IsSubtypeUnionX[t_1, t_2, M'].
11. If t_2 is a union return IsSubtypeXUnion[t_1, t_2, M'].
12. If $t_1 =$ <none> then return #t iff $t_2 =$ <none>.
13. If $t_2 =$ <none> then return #t iff $t_1 =$ <none>.
14. If both t_1 and t_2 are pair classes return IsSubtypePair[t_1, t_2, M']. If only one of t_1 and t_2 is a pair class return #f.
15. If both t_1 and t_2 are procedure types return IsSubtypeGeneralProc[t_1, t_2, M'].
16. If t_1 and t_2 are both vector classes, $t_1 =$ (:vector <a>) and $t_2 =$ (:vector <b>), return IsSubtype[<a>, <b>, M'].
17. If t_1 and t_2 are both value vector classes, $t_1 =$ (:value-vector <a>) and $t_2 =$ (:value-vector <b>), return IsSubtype[<a>, <b>, M'].

18. If both t_1 and t_2 are instances of a parametrized logical type whose type arguments contain type modifiers return `#t` iff the contents of t_1 and t_2 are equal and `#f` otherwise.

19. If t_1 and t_2 are splice expressions return `#t` iff the component type of t_1 is a subtype of the component type of t_2. If only t_2 is a splice expression return `#f`.

20. If t_1 and t_2 are rest expressions return `#t` iff the component type of t_1 is a subtype of the component type of t_2. If only t_2 is a rest expression return `#f`.

21. If t_1 and t_2 are type list expressions return IsGeneralListSubtype$[a,\ b]$ where a and b are the contents of t_1 and t_2 respectively. If only t_2 is a type list expression return `#f`.

22. If t_1 and t_2 are type loop expressions return IsSubtypeLoop$[t_1,\ t_2,\ M']$. If only t_2 is a type loop expression return `#f`.

23. If t_1 and t_2 are type join expressions return IsGeneralListSubtype$[a,\ b]$ where a and b are the contents of t_1 and t_2 respectively. If only t_2 is a type join expression return `#f`.

24. If t_1 and t_2 are instances of a parametrized class return IsSubtypeParam-ClassInst$[t_1,\ t_2,\ M']$.

25. If one of t_1 and t_2 is an instance of a parametrized class and one is class that is not an instance of a parametrized class return IsSubtypeParamClass-Mixed$[t_1,\ t_2,\ M']$.

26. If t_1 and t_2 are classes return IsSubtypeSimple$[t_1,\ t_2]$.

27. else return `#f`.

4.13.2 IsSubtypeSimple

Arguments:

t_1: a class
t_2: another class

Result:

is-subtype? : `#t` if t_1 is a subtype of t_2, `#f` otherwise

Algorithm: IsSubtypeSimple$[t_1,\ t_2]$

1. If $t_1 = t_2$ return `#t` else
2. if $t_2 =$ `<object>` return `#t` else
3. if $t_1 =$ `<object>` and $t_2 \neq$ `<object>` return `#f` else
4. return IsSubtypeSimple$[s,\ t_2]$ where $t_1 ::< s$.

4.13.3 IsGeneralListSubtype

Arguments:
a: a list of type expressions
b: another list of type expressions

Result:
is-subtype? : `#t` iff a is a subtype of b

Algorithm: IsGeneralListSubtype$[a,\ b]$

Let $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_m)$ Return `#t` iff $n = m$ and a_i is a subtype of b_i for all $i = 1, \ldots, n$.

4.13.4 IsSubtypeXUnion

Arguments:
t: a class
u: a union type
M: the set (list) of types already visited

Result:
is-subtype? : `#t` if t is a subtype of u, `#f` otherwise

Algorithm: IsSubtypeXUnion$[t,\ u,\ M]$

1. Let v be the vector of the member types of u, $v :=$ `(`u_1 ...u_n `)` .
2. Let *result* := `#f` .
3. For $i := 1, \ldots, n$
   (a) If IsSubtype$[t,u_i,\ M]$ then set *result* := `#t` and break the loop.
4. Return *result*.

4.13.5 IsSubtypeUnionX

Arguments:
u: a union type
t: a class
M: the set (list) of types already visited

Result:
is-subtype? : `#t` if u is a subtype of t, `#f` otherwise

Algorithm: IsSubtypeUnionX$[u,\ t,\ M]$

1. Let v be the vector of the member types of u, $v :=$ `(`u_1 ...u_n `)` .
2. Let *result* := `#t` .

3. For $i := 1, \dots, n$
   (a) If IsSubtype$[u_i,\ t\ M] =$ `#f` then set *result* := `#f` and break the loop.
4. Return *result*

4.13.6 IsSubtypePair

Arguments:
t: a pair class
u: a pair class
M: the set (list) of types already visited

Result:
is-subtype? : `#t` if t is a subtype of u, `#f` otherwise

Algorithm: IsSubtypePair$[t,\ u,\ M]$

Let `(`a_1 a_2 `)` be the component types of t and `(`b_1 b_2 `)` be the component types of u.

1. If IsSubtype$[a_1,\ b_1,\ M]$ return IsSubtype$[a_2,\ b_2,\ M]$ else return `#f`.

4.13.7 IsSubtypeGeneralProc

Arguments:
t_1: a procedure type
t_2: a procedure type
M: the set (list) of types already visited

Result:
is-subtype? : `#t` iff t is a subtype of u

Algorithm: IsSubtypeGeneralProc$[t_1,\ t_2,\ M]$

If any of the following is true return `#f`:

1. Object t_1 is an abstract procedure type and t_2 is not an abstract procedure type.
2. Object t_1 is a simple procedure class and t_2 is either a parametrized or generic procedure class.
3. Object t_1 is either a parametrized or generic procedure class and t_2 is a simple procedure class.
4. Object t_1 is a parametrized procedure class and t_2 is a generic procedure class.

If some of the following is true:

1. Objects t_1 and t_2 are abstract procedure types.
2. Objects t_1 and t_2 are simple procedure classes.

3. Object t_1 is a simple procedure class and t_2 an abstract procedure type.

return IsSubtypeProc[t_1, t_2, M].

If t_1 is a parametrized procedure class and t_2 is an abstract procedure class return IsSubtypeParamAbstract[t_1, t_2, M]. If t_1 and t_2 are parametrized procedure classes return IsSubtypeParamProc[t_1, t_2]. If t_1 is a generic procedure class and t_2 is an abstract procedure class return IsSubtypeGenAbstract[t_1, t_2, M]. If t_1 and t_2 are generic procedure classes return IsSubtypeGenericProc[t_1, t_2, M].

If t_1 is a generic procedure class and t_2 is a parametrized procedure class return `#t` iff the tree structure of some of the methods of t_1 is identical to t_2(type variables may be named differently).

4.13.8 ProcAttributesMatch

Arguments:

(p_1, a_1, n_1, s_1): attributes of the first procedure
(p_2, a_2, n_2, s_2): attributes of the second procedure

Result:

is-subtype? : `#t` iff the first procedure type can be a subtype of the second

The attributes are: (purity, always returns, never returns, and static method). All of them are boolean valued. The algorithm returns `#t` iff all of the following conditions are true:

- $\neg((\neg p_1) \wedge p_2)$
- $((\neg a_2) \wedge (\neg n_2)) \vee (a_1 = a_2 \wedge n_1 = n_2)$
- $\neg((\neg s_1) \wedge s_2)$

4.13.9 IsSubtypeProc

Arguments:

t: a procedure class
u: a procedure class
M: the set (list) of types already visited

Result:

is-subtype? : `#t` if t is a subtype of u, `#f` otherwise

Algorithm: IsSubtypeProc[t, u, M]

Let A_1 be the procedure attributes of t and A_2 the procedure attributes of u. Let a_1 be the argument list type of t, r_1 the result type of t, and p_1 the purity (boolean value) of t. Define the corresponding variables a_2, r_2, and p_2 for u.

If ProcAttributesMatch[A_1,A_2] is true then

1. Let $st_1 :=$ IsSubtype[a_2, a_1, M] .
2. If $st_1 =$ `#t` then return IsSubtype[r_1, r_2, M] else return `#f`.

else return `#f`.

4.13.10 IsSubtypeParamAbstract

Arguments:

t: a parametrized procedure class
u: an abstract procedure type
M: the set (list) of types already visited

Result:

is-subtype? : `#t` if t is a subtype of u, `#f` otherwise

Algorithm: IsSubtypeParamAbstract[t, u, M]

Let A_1 be the procedure attributes of t and A_2 the procedure attributes of u. If ProcAttributesMatch[A_1,A_2] is true then

1. Deduce type parameters for types t and u. See section 5.10.
2. If some of the type parameters in objects t and u could not be deduced return `#f`.
3. Substitute the deduced type parameter values to objects t and u. Denote the result objects t' and u'. Let a_1 be the argument list type of t' and r_1 the result type of t'. Define the corresponding variables a_2 and r_2 for u'.
4. If r_1 is a subtype of r_2 and a_2 is a subtype of a_1 (note the order) return `#t` else return `#f`.

else return `#f`.

4.13.11 IsSubtypeParamProc

Arguments:

t: a parametrized procedure class
u: a parametrized procedure class

Result:
`#t` if t is identical to u, `#f` otherwise

Algorithm: IsSubtypeParamProc[t, u]

If t and u have the same number of type parameters create new type variables and substitute them into t and u. Return `#t` iff the new type t' is a subtype of the new type u'.

4.13.12 IsSubtypeGenAbstract

Arguments:

t: a generic procedure class
u: an abstract procedure type
M: the set (list) of types already visited

Result:
is-subtype? : #t if t is a subtype of u, #f otherwise

Algorithm: IsSubtypeGenAbstract[t, u, M]

1. Let m := the list of methods of t and n := the number of methods in m .
2. For $i := 1, \ldots, n$
   (a) If IsSubtype[m[i], u, M] break the loop and return #t.
3. Return #f.

4.13.13 IsSubtypeGenericProc

Arguments:
t: a generic procedure class
u: a generic procedure class
M: the set (list) of types already visited

Result:
is-subtype? : #t if t is a subtype of u, #f otherwise

Algorithm: IsSubtypeGenericProc[t, u, M]

1. Let m_1 := the list of methods of t , m_2 := the list of methods of u , n_1 := the number of methods in m_1 , and n_2 := the number of methods in m_2 .
2. Let *result2* := #t .
3. For $i := 1, \ldots, n_1$
   (a) *result1* := #f
   (b) For $j := 1, \ldots, n_2$
      i. If IsSubtype[m_1[i], m_2[j], M] then set *result1* := #t and break the inner loop.
   (c) If *result1* = #f then set *result2* := #f and break the outer loop.
4. Return *result2* .

4.13.14 IsSubtypeParamClassInst

Arguments:
t_1: a class
t_2: another class
M: the set (list) of types already visited

Result:

is-subtype? : `#t` if t_1 is a subtype of t_2, `#f` otherwise

Algorithm: IsSubtypeParamClassInst[t_1, t_2, M]

1. If t_2 = `<object>` return `#t` else
2. if t_1 = `<object>` and $t_2 \neq$ `<object>` return `#f` else
3. if ParamClassInstEqual[t_1, t_2, M] return `#t` else return IsSubtype[s, t_2, M] where t_1 ::< s .

4.13.15 IsSubtypeParamClassMixed

Arguments:

t_1: a class
t_2: another class
M: the set (list) of types already visited

Result:

is-subtype? : `#t` if t_1 is a subtype of t_2, `#f` otherwise

Algorithm: IsSubtypeParamClassMixed[t_1, t_2, M]

1. If t_2 = `<object>` return `#t` else
2. if t_1 = `<object>` and $t_2 \neq$ `<object>` return `#f` else
3. else return IsSubtype[s, t_2, M] where t_1 ::< s .

4.13.16 ParamClassInstEqual

Arguments:

t_1: a class
t_2: another class
M: the set (list) of types already visited

Result:

`#t` if t_1 is equal to t_2, `#f` otherwise

Algorithm: ParamClassInstEqual[t_1, t_2, M]

Let p_1 := `#t` iff t_1 is an instance of a parametrized class and p_2 := `#t` iff t_2 is an instance of a parametrized class .

1. If $(\neg p_1) \wedge (\neg p_2)$ return $t_1 = t_2$ as an object
2. else if $((\neg p_1) \wedge p_2) \vee (p_1 \wedge (\neg p_2))$ return `#f`
3. else if

(a) `(class-of` t_1 `)` is equal to `(class-of` t_2 `)` as an object,

(b) Class t_1 has as many type parameters as class t_2 (we know here that both t_1 and t_2 have to be instances of parametrized classes), and

(c) Each of the type parameter of t_1 is equal to the corresponding type parameter of t_2 (Here equality of types a and b means that $a :< b$ and $b :< a$)

return `#t` else return `#f`.

4.13.17 IsSubtypeLoop

Arguments:

t_1: a loop expression
t_2: another loop expression

Result:

is-subtype? : `#t` iff t_1 is a subtype of t_2

Algorithm: IsSubtypeLoop[t_1, t_2]

If the iteration variables of t_1 and t_2 are the same return `#t` iff the subtype lists of t_1 and t_2 are equal (have equal tree structures) and the iteration expression of t_1 is a subtype of the iteration expression of t_2. If the iteration variables are not equal create a new type variable, substitute it into t_1 and t_2, and do the same check as above.

4.13.18 IsSubtypeXSignature

Arguments:

t_1: a type that is not a signature
t_2: a signature type
M: the set (list) of types already visited

Result:

is-subtype? : `#t` iff t_1 is a subtype of t_2

Algorithm: IsSubtypeXSignature[t_1, t_2, M]

We have $t_1 :< t_2$ iff for each specifier $s =$ `(`*proc-name args result attributes* `)` in the complete specifier list of t_2 there exists a procedure (simple, parametrized, or generic) with name *proc-name* so that the class of this procedure is a subtype of the abstract procedure type `(:procedure` *args result attributes* `)` where the keyword `this` has been substituted with type t_1. We will use this algorithm for computing the subtyping of parametrized signatures, too.

4.13.19 IsSignatureSubtype

Arguments:

t_1: a signature type
t_2: a signature type

M: the set (list) of types already visited

Result:

is-subtype? : `#t` iff t_1 is a subtype of t_2

Algorithm: IsSignatureSubtype[t_1, t_2, M]

We have t_1 :< t_2 iff for each specifier s_2 = `(`*proc-name*$_2$ *args*$_2$ *result*$_2$ *attributes*$_2$ `)` in the complete specifier list of t_2 there exists a specifier s_1 = `(`*proc-name*$_1$ *args*$_1$ *result*$_1$ *attributes*$_1$ `)` in the complete specifier list of t_1 so that the procedure names *proc-name*$_1$ and *proc-name*$_2$ are equal and `(:procedure` *args*$_1$ *result*$_1$ *attributes*$_1$ `)` is a subtype of `(:procedure` *args*$_2$ *result*$_2$ *attributes*$_2$ `)`.

4.14 Algorithms to Compute Equivalence of Objects

4.14.1 General

When we refer to Scheme procedures in this section we assume that they behave as specified in [3]. Note that algorithms EqualPrimitiveValues? and EqualPrimitiveObjects? differ only in their handling of strings.

4.14.2 Equal?

Arguments:

obj1: an object
obj2: an object
v: the set (list) of object pairs already visited

Result:

`#t` if *obj1* is equal to *obj2*, `#f` otherwise

Algorithm: Equal?[*obj1*, *obj2*, v]

1. If *obj1* and *obj2* are the same nonprimitive object return `#t`.
2. If (*obj1 obj2*) $\in v$ return `#t`.
3. Let *cl1* to be the class of *obj1* and *cl2* the class of *obj2*.
4. If not EqualTypes?[*cl1*, *cl2*, *visited*] return `#f`.
5. If *obj1* (and *obj2*) is a primitive object return EqualPrimitiveValues?[*obj1*, *obj2*].
6. Let v'`:= (cons (cons` *obj1 obj2*`)` v`)` .
7. If *obj1* (and *obj2*) is a pair return EqualPairs?[*obj1*, *obj2*, v'].
8. If *cl1* (and *cl2*) is a type return EqualTypes?[*obj1*, *obj2*, v'].

9. If *cl1* = `(:value-vector` t `)` or *cl1* = `(:mutable-value-vector` t `)` for some type t return EqualVectors?[*obj1*, *obj2*, v'].
10. If *cl1* (and *cl2*) is equal by value return EqualByValue?[*obj1*, *obj2*, v'].
11. Otherwise return `#f`.

4.14.3 EqualContents?

Arguments:

obj1: an object
obj2: an object
v: the set (list) of object pairs already visited

Result:

`#t` if the contents of *obj1* are equal to the contents of *obj2*, `#f` otherwise

Algorithm: EqualContents?[*obj1*, *obj2*, v]

1. If *obj1* and *obj2* are the same nonprimitive object return `#t`.
2. If (*obj1 obj2*) $\in v$ return `#t`.
3. Let *cl1* to be the class of *obj1* and *cl2* the class of *obj2*.
4. If not EqualTypes?[*cl1*, *cl2*, *visited*] return `#f`.
5. If *obj1* (and *obj2*) is a primitive object return EqualPrimitiveValues?[*obj1*, *obj2*].
6. Let v'`:= (cons (cons` *obj1* *obj2*`)` v`)` .
7. If *obj1* (and *obj2*) is a pair return EqualPairContents?[*obj1*, *obj2*, v'].
8. If *cl1* (and *cl2*) is a type return EqualTypes?[*obj1*, *obj2*, v'].
9. If *cl1* (and *cl2*) is a general vector class return EqualVectorContents?[*obj1*, *obj2*, v'].
10. Otherwise return EqualFields?[*obj1*, *obj2*, v'].

4.14.4 EqualObjects?

Arguments:

obj1: an object
obj2: an object

Result:

`#t` if *obj1* and *obj2* are the same object, `#f` otherwise

Algorithm: EqualObjects?[*obj1*, *obj2*]

1. If *obj1* is a primitive object
   (a) Let *cl1* be the class of *obj1* and *cl2* the class of *obj2*.
   (b) If EqualTypes?[*cl1*, *cl2*, ()]
      i. return EqualPrimitiveObjects?[*obj1*, *obj2*]
   else
      i. return `#f`

   else

   (a) If *obj1* and *obj2* are the same nonprimitive object return `#t` else return `#f`.

4.14.5 EqualTypes?

Arguments:

t1: a type
t2: a type
v: the set (list) of type pairs already visited

Result:

`#t` if types *t1* and *t2* are equal, `#f` otherwise

Algorithm: EqualTypes?[*t1*, *t2*, *v*]

1. If *t1* and *t2* are the same nonprimitive object return `#t`.
2. If $(t1\ t2) \in v$ return `#t`.
3. Let v'`:= (cons (cons` *t1* *t2*`)` *v*`)` .
4. If *t1* and *t2* are pair classes return

$$\text{EqualTypes?}[t1[1], t2[1], v'] \wedge \text{EqualTypes?}[t1[2], t2[2], v'].$$

5. If *t1* or *t2* is a pair class return `#f`.
6. If *t1* is a general vector class then
   (a) If the classes of *t1* and *t2* are not the same nonprimitive object return `#f`.
   (b) Let *u1* be the component type of *t1* and *u2* the component type of *t2*.
   (c) Return EqualTypes?[*u1*, *u2*, v'].
7. If *t1* and *t2* are classes
   (a) If *t1* and *t2* are the same nonprimitive object return `#t` else return `#f`.
8. If *t1* or *t2* is a class return `#f`.
9. Otherwise return $t1 :< t2 \wedge t2 :< t1$.

4.14.6 EqualByValue?

Arguments:

obj1: an object
obj2: an object
v: the set (list) of object pairs already visited

Result:

equal? : `<boolean>`

Algorithm: EqualByValue?[*obj1*, *obj2*, *v*]

Note: We assume that the classes of *obj1* and *obj2* are equal in the sense of EqualTypes?.

1. Let *cl* be the class of *obj1* (and *obj2*).
2. Let *result* := `#t` .
3. For each field *fld* in the field list of class *cl* do
   (a) Let *f1* := the value of the field *fld* in object *obj1* and *f2* := the value of the field *fld* in object *obj2*.
   (b) If not Equal?[*f1*, *f2*, *v*] set *result* := `#f` and break the loop.
4. Return *result*.

4.14.7 EqualFields?

Arguments:

obj1: an object
obj2: an object
v: the set (list) of object pairs already visited

Result:

equal? : `<boolean>`

Algorithm: EqualFields?[*obj1*, *obj2*, *v*]

Note: We assume that the classes of *obj1* and *obj2* are equal in the sense of EqualTypes?.

1. Let *cl* be the class of *obj1* (and *obj2*).
2. Let *result* := `#t` .
3. For each field *fld* in the field list of class *cl* do
   (a) Let *f1* := the value of the field *fld* in object *obj1* and *f2* := the value of the field *fld* in object *obj2*.
   (b) If not EqualContents?[*f1*, *f2*, *v*] set *result* := `#f` and break the loop.
4. Return *result*.

4.14.8 EqualPairs?

Arguments:
p1: a pair
p2: a pair
v: the set (list) of object pairs already visited

Result:
Equal?[*p1*[1], *p2*[1], *v*] ∧ Equal?[*p1*[2], *p2*[2], *v*]

4.14.9 EqualPairContents?

Arguments:
p1: a pair
p2: a pair
v: the set (list) of object pairs already visited

Result:
EqualContents?[*p1*[1], *p2*[1], *v*] ∧ EqualContents?[*p1*[2], *p2*[2], *v*]

4.14.10 EqualPrimitiveValues?

Arguments:
obj1: a primitive value
obj2: a primitive value

Result:
equal? : `<boolean>`

Preconditions:
The classes of *obj1* and *obj2* have to be equal and they have to be a primitive class.

Algorithm: EqualPrimitiveValues?[*obj1*, *obj2*]

1. If *obj1* is a `<boolean>` or `<symbol>` value return the result of Scheme expression `(eq?` *obj1* *obj2*`)`
2. If *obj1* is an integer or a real number return the result of Scheme expression `(=` *obj1* *obj2*`)`.
3. If *obj1* = `null` return `#t`.
4. If *obj1* is a `<eof>` value return `#t`.
5. If *obj1* is a `<character>`, `<input-port>`, or `<output-port>` value return the result of Scheme expression `(eqv?` *obj1* *obj2*`)`
6. If *obj1* is a `<string>` value return the result of Scheme expression `(string=?` *obj1* *obj2*`)`

4.14.11 EqualPrimitiveObjects?

Arguments:
obj1: a primitive value
obj2: a primitive value

Result:
equal? : `<boolean>`

Preconditions:
The classes of *obj1* and *obj2* have to be equal and they have to be a primitive class.

Algorithm: EqualPrimitiveObjects?[*obj1*, *obj2*]

1. If *obj1* is a `<boolean>` or `<symbol>` value return the result of Scheme expression `(eq?` *obj1* *obj2*`)`
2. If *obj1* is an integer or a real number return the result of Scheme expression `(=` *obj1* *obj2*`)`.
3. If *obj1* = `null` return `#t`.
4. If *obj1* is a `<eof>` value return `#t`.
5. If *obj1* is a `<character>`, `<string>`, `<input-port>`, or `<output-port>` value return the result of Scheme expression `(eqv?` *obj1* *obj2*`)`

4.14.12 EqualVectors?

Arguments:
v1: a general vector
v2: a general vector
v: the set (list) of object pairs already visited

Result:
`#t` if the lengths of *v1* and *v2* are equal and all the elements of *v1* and *v2* are equal in the sense of Equal?, `#f` otherwise

4.14.13 EqualVectorContents?

Arguments:
v1: a general vector
v2: a general vector
v: the set (list) of object pairs already visited

Result:
`#t` if the lengths of *v1* and *v2* are equal and all the elements of *v1* and *v2* are equal in the sense of EqualContents?, `#f` otherwise

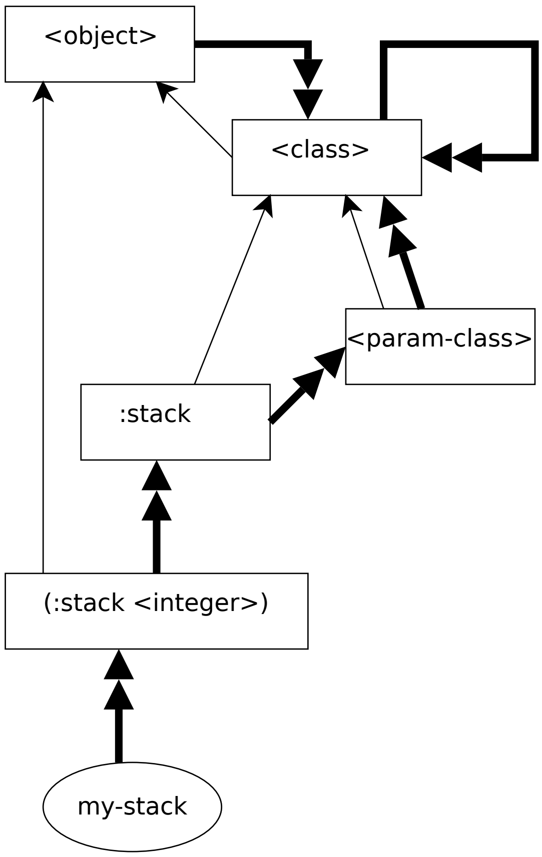

Figure 4.2: Example inheritance hierarchy for parametrized classes.

Chapter 5

Procedures

5.1 General

Theme-D has three kinds of procedures: *simple procedures*, *generic procedures*, and *parametrized procedures*. A procedure is applied with syntax

(*proc* `arg-1` ...`arg-n`)

where *proc* is the procedure and `arg-1`, ..., `arg-n` are the arguments passed to *proc*. It is possible for a procedure to have no arguments. As regards the simple procedures *proc* can be any expression that returns a simple procedure. See section 6.14.1 in this manual and `define-simple-method` and `define-param-proc` in chapter 3 of the standard library reference for procedure definition syntax. If a procedure defines a *rest argument* an arbitrary number of instances of the rest argument type can be passed to the procedure at the end of the argument list.

Procedures should only be applied in procedure bodies. In particular, a defining expression of a toplevel definition must not be a procedure application. As an exception to this rule applications of the procedure `list` generated by quasiquotation are legal toplevel, too.

Every expression in Theme-D is either *pure* or *nonpure*. A pure expression cannot have any side effects (or should not have in case of `force-pure` and `force-pure-expr`). An application of a pure procedure is a pure expression. Other procedures are nonpure. A procedure may also be declared `force-pure`, in which case the procedure may contain nonpure expressions but the Theme-D compiler and linker handle the procedure as pure. Note that the purity of a procedure is not necessarily the same as the purity of a procedure application calling the procedure. If the procedure is pure and some of the subexpressions of the procedure application is nonpure the procedure application expression is nonpure. The implementation of a pure procedure may change the internal variables of the procedure. More formally, it is legal to change mutable variables defined inside the nearest lexically enclosing pure procedure implementation of the expression changing the variable. A typical use of this feature is a pure procedure having loop variables.

A procedure may also be declared to *never returning* or *always returning*.

An example of a procedure returning never is `raise`, which raises an exception. A procedure returning always may not raise any exceptions and it has to handle any exceptions being generated in it. When a procedure is neither always returning or never returning we say that it *may return*.

If a procedure is a method in some generic procedure the procedure may be declared *static*. If a method is static it is dispatched statically, i.e. when the method is selected in compile-time dispatch that method is used instead of doing runtime dispatch.

A lambda expression or a parametrized lambda expression may be optionally assigned a name. This name is used for debugging purposes (runtime backtraces) only. If you define a variable having a lambda expression (or a parametrized lambda expression) value with a define or let expression the lambda expression is assigned the name of the variable automatically and you do not have to specify it in the lambda expression.

5.2 Simple Procedures

Every simple procedure is an instance of some *simple procedure class*. A simple procedure class is an instance of the parametrized class `:simple-proc`. See subsection 6.5 for the syntax for defining procedure classes. When you define a procedure the Theme-D compiler and/or linker deduce the procedure class from the procedure argument list, result type, and purity specifier automatically, though.

5.3 Generic Procedures

A generic procedure is a collection of simple or parametrized procedures, which are called methods. When a generic procedure is called and an argument list is passed to it Theme-D first checks which of the methods of the generic procedure can be called with the argument list, i.e. the type af the argument list is a subtype of the method argument list type. Theme-D then finds out which of the suitable methods is the best match. If a unique best match is not found an exception is raised. These checks occur generally run-time. The dispatch of a generic procedure application must succeed statically even though the static dispatch is allowed to be ambiguous. No two methods of the same generic procedure may have identical argument list types.

Suppose that a generic procedure has two distinct methods having argument list types A and B and result types R and S, respectively. If A is a subtype of B then R has to be a subtype of S. This is called the *covariant typing rule*. The covariant typing rule allows Theme-D to deduce a supertype of the result type of a generic procedure application at compile time.

When a Theme-D program is linked all the generic procedure with same name will be merged and the methods defined for each of the merged generic procedures will be put into the new generic procedure.

5.4 Parametrized Procedures

A parametrized procedure is a procedure having type parameters. When these type parameters are assigned type values we get a simple procedure. Note that a parametrized procedure is not a simple procedure itself. The values of the type parameters are usually not specified explicitly. Theme-D deduces the values from the argument types of a parametrized procedure application. This is done in translation time. If Theme-D is unable to deduce the type parameter values a translation error is signalled.

5.5 Abstract Procedure Types

Abstract procedure types are instances of metaclass `:procedure`. An object whose static type is a abstract procedure type may be any kind of procedure, i.e. simple, generic, or parametrized procedure, with a proper class.

5.6 Subtyping of Procedure Types

A simple or abstract procedure class A is a subtype of a simple or abstract procedure class B if and only if

- One of the following is true:
  - Objects A and B are abstract procedure types.
  - Objects A and B are simple procedure classes.
  - Object A is a simple procedure class and B an abstract procedure type.
- The argument list type of B is a subtype of the argument list type of A (note the order).
- The result type of A is a subtype of the result type of B.
- Either A and B are both pure, both nonpure, or A is pure and B nonpure.
- Either B may return or the returning attributes of A and B are equal.

See subsections 4.13.9, 4.13.10, 4.13.11, 4.13.12, and 4.13.13 for further information on procedure type subtyping.

5.7 Argument Type Modifiers and Static Type Expressions

The type-valued expressions in Theme-D may contain several *argument type modifiers*. These modifiers are

splice Adds the arguments of **splice** into the enclosing type list definition.

rest Specifies the component type for the variable argument list part of the procedure that is being defined.

type-loop Assigns the loop variable with the expressions from the list given and binds the loop expression with each value.

join-tuple-types Concatenates the elements of all component types.

Expression

```
(:tuple a1 a2 ...  (splice (:tuple b1 b2 ...))  c1 c2 ...)
```

is equivalent to

```
(:tuple a1 a2 ...  b1 b2 ...  c1 c2 ...)
```

Expression type **splice** is mainly intended to be used with **type-loop** expressions.

Expression

```
(type-loop %itervar values expression)
```

will iterate type variable `%itervar` in the type list `values`. Type variable `%itervar` is bound to a value from `values` and expression `expression` is evaluated with this binding at each iteration. The result value of the **type-loop** expression is a type list containing the evaluated expressions. A type variable whose value is a type list shall be accepted as the argument list type for `:procedure`.

Example:

```
(define-param-proc map (%arglist %return-type)
  (prim-proc map
    ((:procedure ((splice %arglist)) %return-type pure)
    (splice (type-loop %iter %arglist (:uniform-list %iter))))
    (:uniform-list %return-type)
    pure))
```

When P is a procedure with declared argument types a_1, ..., a_n the *argument type list descriptor* of P is defined to be (a_1 ...a_n).

A *static type expression* is defined as follows:

- Every constant whose value is a static type expression is a static type expression.
- Every constant whose value is a type is a static type expression.
- Every instantiation of a parametrized type is a static type expression. The type parameters have to be static type expressions.
- A `:tuple` expression is a static type expression. The type parameters have to be static type expressions.
- Every valid application of an argument type modifier is a static type expression.

- Every type variable is a static type expression.

5.8 Algorithm to Deduce the Values of Argument Variables

5.8.1 TranslateArguments

This algorithm deduces the values of procedure argument variables from the arguments in procedure application. When l is a list we define $N(l)$ to be the length of the list l.

Arguments:
a_1 ... a_n : argument descriptors
v_1 ... v_m : argument values in the procedure application

Result:
w_1 ... w_n : values assigned to each argument variable

Algorithm: TranslateArguments[a_1, ..., a_n, v_1, ..., v_m]

1. If $n = 0 \wedge m \neq 0$
2. then raise error
3. else
   (a) $c := (v_1 \ldots v_m)$
   (b) $r := ()$
   (c) For $i = 1, \ldots, n$
      i. $t :=$ TranslateArgument[a_i,c]
      ii. $r :=$ Concatenation of r and $t[1]$
      iii. $c := t[2]$
   (d) Return r.

5.8.2 TranslateArgument

Arguments:
a : The argument descriptor being handled
c : The application arguments left

Result:
A pair whose first element is the value/list of values to be assigned to the argument a and the second element a list of the application arguments left after handling the current argument translation

Algorithm: TranslateArgument[a, c]

If any of the following is true:

- a is a type
- a is a list of static type expressions
- a is a type join expression

return $((c_1\ )(c_2\ ...c_{N(c)}\ ))$. If $N(c) = 0$ in the case above raise error.
If $a = (\textbf{rest}\ r\ )$ return $((c\ )())$.
Suppose that $a = (\textbf{splice}\ s\ )$. Now s has to be a list of static type expressions. Define $l := N(s)$. If $N(c) \geq l$ return $(((c_1\ ...c_l\ ))(c_{l+1}\ ...c_{N(c)}\ ))$.

5.9 Algorithm to Dispatch Generic Procedure Applications

5.9.1 SelectBestMatch

Arguments:

$l = (t_1\ ...t_n\ )$: Call arguments
$s_j = (s_{j,1}\ ...s_{j,p(j)}\ r_j\ )$: Declared method argument lists, $j = 1, \ldots, m$

Result:
result: Either found, ambiguous, or not found.
methods: The dispatched methods found.

Algorithm: SelectBestMatch$[l,\ s_1,\ ...\ s_m]$

1. Deduce the type parameters for all the parametrized methods. Reject all the methods for which all type parameters could not be deduced.
2. Set v to a vector of m elements, each of which equal to `#t`.
3. Set $v[i] :=$ `#f` iff $\neg\ l :< s_i$.
4. For each $i = 1, \ldots, n$
   (a) Define
   $$a(j,i) := \begin{cases} s_{j,i}; & i \leq p(j) \\ r_j; & i > p(j) \end{cases}$$
   and set w a vector of m elements with value `#f`.
   (b) For each $j = 1, \ldots, m$: if $v[j] = \#t$ and $a(j,i) :< t_i$ set $w[j] := \#t$.
   (c) If $w[j_0] = \#t$ for some $j_0 \in \{1, \ldots, m\}$ then set $v := w$ else do SelectNearestMethod$[i,\ v,\ l,\ s_1,\ ...,\ s_m]$
   (d) If there is one or zero j for which $v[j] = \#t$ break the loop.
5. If $v[j] =$ `#t` for exactly one $j \in \{1, \ldots, m\}$ then the result of the algorithm is "found" and the result method is method number j. If $v[j] =$ `#t` for more than one $j \in \{1, \ldots, m\}$ the result of the algorithm is "ambiguous" and the result methods are all the methods for which $v[j] =$ `#t`. If $v[j] =$ `#f` for all $j \in \{1, \ldots, m\}$ the result of the algorithm in "not found".

5.9.2 SelectNearestMethods

Arguments:

i : Index to the argument list
v : Boolean values marking the methods included in computation
$l = (t_1 \ldots t_n)$: Same as in SelectBestMatch
$s_j = (s_{j,1} \ldots s_{j,p(j)}\ r_j)$: Same as in SelectBestMatch

Result:
v : Boolean values marking the methods included in computation

Algorithm: SelectNearestMethods[i, v, l, s_1, ..., s_m]

Define $a(j, i)$ as in algorithm SelectBestMethod.

For each $j = 1, \ldots, m$
For each $k = 1, \ldots, m$
If $j \neq k$, $v[j] \wedge v[k]$, and $a(j, i) :< a(k, i)$ set $v[k] := \#f$.

5.10 Algorithm to Dispatch Parametrized Procedure Applications

This algorithm computes only suggestions for the type parameter values of a given parametrized procedure. We will bound the type variables in the type of the parametrized procedure with the suggestions. Then we will check that the type of the application argument list is a subtype of the bound parametrized procedure type.

When we compile an implementation of a parametrized procedure we fix the type parameters of the procedure so that their values are not deduced and the other type variables may be represented in terms of them.

5.10.1 DeduceArgumentTypes

Arguments:

src: A static type expression
target: A static type expression
T: An object containing type variable bindings
F: A list of fixed type variables

Result:
all-found? : `#t` iff values were found for all the nonfixed type variables in *src* and *target*

Algorithm: DeduceArgumentTypes[*src*, *target*, T, F]

1. Set *state* := 2 , *old-count-source* := 0 , *old-count-target* := 0 , *cur-src* := *src* , *cur-target* := *target* , and *dir-forward?* := `#t` .
2. Until *state* ≤ 0 do

(a) If *dir-forward?*

i. If $state > 0$

A. Apply algorithm DeduceStepForward[$cur\text{-}src$, $cur\text{-}target$, T, F, $old\text{-}count\text{-}target$, $state$] and store the result into res.

B. Set $state := res[1]$ and $old\text{-}count\text{-}target := res[2]$.

C. Bind all the bindings of T in expression *target* and store the result into *cur-target*.

D. Bind all the bindings of T in expression *src* and store the result into *cur-src*, else

i. If $state > 0$

A. Apply algorithm DeduceStepForward[$cur\text{-}src$, $cur\text{-}target$, T, F, $old\text{-}count\text{-}src$, $state$] and store the result into res.

B. Set $state := res[1]$ and $old\text{-}count\text{-}src := res[2]$.

C. Bind all the bindings of T in expression *src* and store the result into *cur-src*.

D. Bind all the bindings of T in expression *target* and store the result into *cur-target*.

(b) Set *dir-forward?* to ¬*dir-forward?*.

3. Return `#t` iff $state = -1$.

5.10.2 DeduceStepForward

Arguments:

src: A static type expression
target: A static type expression
T: An object containing type variable bindings
F: A list of fixed type variables
old-count: The number of type variables already deduced
old-state: The old state of the algorithm

Result:
new-state: The new state of the algorithm
new-count: The number of type variables deduced

Algorithm: DeduceStepForward[src, $target$, T, F, $old\text{-}count$, $old\text{-}state$]

1. Apply algorithm DeduceTypeParams[src, $target$, T, F, ()].

2. Set $new\text{-}count :=$ the number of bindings in T .

3. If all the type variables in *src* and *target* were deduced return `(-1` *new-count* `)`.

4. If $old\text{-}count = new\text{-}count$ let $s := old\text{-}state$ - 1 and return `(`*s new-count* `)`.

5. Otherwise return `(2` *new-count* `)`.

5.10.3 DeduceStepBackward

Arguments:

src: A static type expression
target: A static type expression
T: An object containing type variable bindings
F: A list of fixed type variables
old-count: The number of type variables already deduced
old-state: The old state of the algorithm

Result:
new-state: The new state of the algorithm
new-count: The number of type variables deduced

Algorithm: DeduceStepBackward[*src*, *target*, T, F, *old-count*, *old-state*]

1. Apply algorithm DeduceTypeParams[*target*, *src*, T, F, ()].
2. Set *new-count* := the number of bindings in T .
3. If all the type variables in *src* and *target* were deduced return `(-1` *new-count* `)`.
4. If *old-count* = *new-count* let s := *old-state* - 1 and return `(`s *new-count* `)`.
5. Otherwise return `(2` *new-count* `)`.

5.10.4 DeduceTypeParams

Arguments:

src: A list of list of static type expressions
dest: A static type expression
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceTypeParams[*src*, *dest*, T, F, v]

1. If *src* is not a (general) pair and *dest* is a splice expression compute DeduceTypeParams[(*src*), c, T, F, *v2*] where c is the component type of *dest*. If the condition above does not hold and *src* is not a pair return `#f` (this may be an error situation).
2. Let x be the head of list *src*. If pair (x,*dest*) is contained in v return. Otherwise add (x,*dest*) into *v2*.
3. Set *src2* to be the value of PrepareSourceType[*src*].

4. If *src2* is not a pair or there are not any type variables in *dest* return.

5. If *dest* is a type variable compute DeduceSimpleType[*src2*, *dest*, T, F, *v2*] and return.

6. If *src* and *dest* are both signatures compute DeduceSgnSgn[*src*, *dest*, T, F, *v2*] and return.

7. If *dest* is a signature and *src* is not a signature compute DeduceNotSgnSgn[*src*, *dest*, T, F, *v2*] and return.

8. If *dest* is not a signature and *src* is a signature return.

9. If *dest* is a pair or a pair class compute DeducePair[*src2*, *dest*, T, F, *v2*] and return.

10. If *dest* is a rest expression compute DeduceRest[*src2*, *dest*, T, F, *v2*] and return.

11. If *dest* is a splice expression compute DeduceSplice[*src2*, *dest*, T, F, *v2*] and return.

12. If *dest* is a type loop expression compute DeduceTypeLoop[*src2*, *dest*, T, F, *v2*] and return.

13. If *src2*[1] is a union and *dest* is not a union compute DeduceUnionX[*src2*[1], *dest*, T, F, *v2*] and return.

14. If *src2*[1] is not a union and *dest* is a union compute DeduceXUnion[*src2*, *dest*, T, F, *v2*] and return.

15. If *src2*[1] is a union and *dest* is a union compute DeduceUnionUnion[*src2*[1], *dest*, T, F, *v2*] and return.

16. If *src2*[1] is a generic procedure class and *dest* is an abstract procedure type compute DeduceGenAbst[*src2*[1], *dest*, T, F, *v2*] . and return.

17. If *dest* and *src2* are parametrized type instances whose type parameters contain type modifiers return `#t` if the types and type parameters of these instances are equal. If only one of *dest* and *src2* is this kind of instance return `#f`. Note that type modifiers may be optimized away by the Theme-D compiler and linker in which case these conditions are not fulfilled.

18. Compute DeduceSubexprs[*src2*, *dest*, T, F, *v2*] and return.

5.10.5 PrepareSourceType

Arguments:
t: a list of static type expressions

Result:
u : a modified list of static type expressions

Algorithm: PrepareSourceType[t]

Let $t = (t_1, \ldots, t_n)$. Define

$$r := \begin{cases} c; & \text{if } t_1 \text{ is a splice expression} \\ t_1; & \text{otherwise} \end{cases}$$

where c is the component type of t_1. Return $(r, t_2, \ldots, t_n)$.

5.10.6 DeduceSubexprs

Arguments:

src: A list of list of static type expressions
dest: A static type expression
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceSubexprs[*src*, *dest*, T, F, v]

1. Set *comp* := The component list of the head of *src* .
2. Set *src-new* := A list whose head is *comp* and whose tail is the tail of *src* .
3. Compute DeduceTypeParams[*src-new*, *subexprs2*, T, F, v].

5.10.7 DeduceSimpleType

Arguments:

src: A list of list of static type expressions
dest: A variable reference to a type
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceSimpleType[*src*, *dest*, T, F, v]

1. Let *var* be the variable which *dest* refers to.
2. If *src* is a list, *var* is a type variable, and *var* is not already contained in T then add the binding of *var* with the head of *src* into T.
3. If the formerly deduced type variables contain variable *var* substitute the new binding into them.
4. If the new deduced value contains formerly deduced type variables substitute them into the new value.

5.10.8 DeducePair

Arguments:

src: A list of list of static type expressions
dest: A pair or a pair class
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeducePair[*src*, *dest*, T, F, v]

1. Let *src2* := the head of the pair *src*
2. Let u := the head of the pair *dest* and compute DeduceTypeParams[*src2*, u, T, F, v].
3. Let r := (*r0*) where *r0* := the tail of the pair *src2* and s := the tail of the pair *dest* and compute DeduceTypeParams[r, s, T, F, v].

5.10.9 DeduceRest

Arguments:

src: A list of list of static type expressions
dest: A rest expression
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceRest[*src*, *dest*, T, F, v]

Let t be the component type of the rest expression *dest*. Compute DeduceTypeParams[*src*, t, T, F, v] .

5.10.10 DeduceSplice

Arguments:

src: A list of list of static type expressions
dest: A splice expression
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceSplice[*src*, *dest*, T, F, v]

Let t be the component type of the rest expression *dest*. Let l be the single element list containing *src*. Compute DeduceTypeParams$[l,\ t,\ T,\ F,\ v]$.

5.10.11 DeduceTypeLoop

Arguments:

src: A list of list of static type expressions
dest: A type loop expression
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceTypeLoop$[src,\ dest,\ T,\ F,\ v]$

Let *iter-var* be the iteration variable of *dest*, *iter-expr* the iteration expression of *dest*, and *subtype-list* the subtype list of *dest*.

1. Let *source-list* be the first element of *src*. If *source-list* is not a list raise error else we have *source-list* = `(`t_1 ...t_n `)`.
2. Let *guessed-items* to be the empty list.
3. If *source-list* is not empty then
   (a) For $i = 1, \ldots, n$
      i. Let U be a copy of T sharing the same contents.
      ii. Call DeduceTypeParams$[t_i,\ iter\text{-}expr,\ U,\ F,\ v]$.
      iii. If U contains a binding for type variable *iter-var* add the binding into the list *guessed-items*.
   (b) If *guessed-items* contains `#f` return.
   (c) If *subtype-list* is a type variable u and u is not already contained in T then
      i. Let b be the list consisting of the tails of the pairs in *guessed-items* with the same order.
      ii. Add a binding of u with b into T.
   (d) Denote the elements of *guessed-items* with g_j, $j = 1, \ldots, m$. For $j = 1, \ldots, m$
      i. Construct list *bindings* by appending the contents of T and g_j.
      ii. Create list r by applying *bindings* in expression *iter-expr*.
      iii. Set $h_j := r$.
   (e) Compute DeduceTypeParams$[src,\ h,\ T,\ F,\ v]$.
4. If *subtype-list* is a variable reference to a type variable and T does not contain a binding of *subtype-list* add a binding of *subtype-list* with the empty list into T.

5.10.12 DeduceUnionX

Arguments:

src: A union expression
dest: A static type expression
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceUnionX[*src*, *dest*, T, F, v]

Let u_1, ..., u_n be the member types of union *src*. For $i = 1, \ldots, n$ compute DeduceTypeParams[(u_i), *dest*, T, F, v] .

5.10.13 DeduceXUnion

Arguments:

src: A list of static type expressions
dest: A union expression
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceXUnion[*src*, *dest*, T, F, v]

Let u_1, ..., u_n be the member types of union *dest*. For $i = 1, \ldots, n$ call DeduceTypeParams[*src*, u_i, T, F, v] .

5.10.14 DeduceUnionUnion

Arguments:

src: A union expression
dest: A union expression
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceXUnion[*src*, *dest*, T, F, v]

Let t_1, ..., t_m be the member types of union *src* and u_1, ..., u_n the member types of union *dest*. Let $p := \min\{m, n\}$. For $i = 1, \ldots, p$ compute DeduceTypeParams[(t_i), u_i, T, F, v] .

5.10.15 DeduceGenAbst

Arguments:

t1: A generic procedure class
t2: An abstract procedure type
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceGenAbst[*t1*, *t2*, T, F, v]

1. If *t1* contains type variables compute DeduceGenAbstArgList[*t1*, *t2*, T, F, v]
2. If *t2* contains type variables compute DeduceGenAbstResult[*t1*, *t2*, T, F, v]

5.10.16 DeduceGenAbstResult

Arguments:

t1: A generic procedure class
a: The target argument list type
r: The target result type
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceGenAbstResult[*t1*, a, r, T, F, v]

1. Let m be the method class list of *t1*. Compute *result* and *method* with algorithm SelectBestMatch[a, m].
2. If an unambiguous match was found let b be the result type of m and apply algorithm DeduceTypeParams[(b), r, T, F, v].

5.10.17 DeduceGenAbstArgList

Arguments:

t1: A generic procedure class
a: The target argument list type
r: The target result type
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceGenAbstArgList[$t1$, a, r, T, F, v]

1. Let m be the method class list of $t1$. Compute *result* and *method* with algorithm SelectBestMatch[a, m].
2. If an unambiguous match was found let c be the argument list type of m and apply algorithm DeduceTypeParams[(c), a, T, F, v].

5.10.18 DeduceNotSgnSgn

Arguments:

src: A static type expressions
dest: A signature
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceNotSgnSgn[*src*, *dest*, T, F, v]

For each procedure specifier s in *dest* define p be the type of the corresponding procedure and define q by substituting `this` by *src* in s. Apply algorithm DeduceTypeParams[(p), q, T, F, v].

5.10.19 DeduceSgnSgn

Arguments:

src: A signature
dest: A signature
T: An object containing type variable bindings
F: A list of fixed type variables
v: A set (list) of expression pairs visited

No result value.

Algorithm: DeduceSgnSgn[*src*, *dest*, T, F, v]

For each procedure specifier p in signature *src*
  for each procedure specifier q in signature *dest*
    If the names of p and q are equal apply algorithm
    DeduceTypeParams[(p), q, T, F, v].

Chapter 6

Expressions

6.1 General

Note that many of the control structures in Theme-D are defined by the standard library. See the standard library reference for these. If the type of a syntax variable (printed in italic) is not defined it is assumed to be an expression. If we make an union of a set of types and some of these types is `<none>` the union is also `<none>`. Syntax element "identifier" means a legal Theme-D identifier. Syntax element "null" means an empty list, denoted by either `null` or `()`.

6.2 Macros

Theme-D has a hygienic and lexically scoped macro system similar to Scheme macros. The keywords `define-syntax`, `let-syntax`, `letrec-syntax`, and `syntax-case` are defined for the macro system. The macro system is partly implemented by the Theme-D standard library. Some of the Theme-D control structures are implemented by macros in the standard library. Macros cannot be declared. When you want to export macros you have to put them into the interface file of a module. See chapter 3 in Theme-D Standard Library Reference and Scheme standard documentation [3] for more information.

The macro transformers must expand to a special macro transformer language resembling Scheme. The value returned by a macro transformer has to be a Theme-D expression.

6.2.1 Forms in the Macro Transformer Language

The following forms are built-in:

- **$lambda**
- **$let**
- **if-object**
- **if**
- **begin**

- **set!**
- **quote**

The following forms are implemented by the Theme-D standard library:

- **$let***
- **$letrec**
- **$letrec***
- **$and**
- **$or**

The keywords starting with '$' behave like the corresponding keywords in Scheme. The other keywords behave like the corresponding keywords in Theme-D.

6.2.2 Procedures in the Macro Transformer Language

These procedures work as the corresponding procedures without the leading '$' in Scheme, see [3]:

- `$cons`
- `$car`
- `$cdr`
- `$pair?`
- `$null?`
- `$list?`
- `$list`
- `$for-all`
- `$map`
- `$apply`
- `$equal?`
- `$=`
- `$>=`
- `$>`
- `$length`
- `$append`
- `$+`

- `$-`
- `$vector`
- `$vector->list`
- `$raise`

These procedures are defined by the SRFI-72 implementation (without the leading '$'):

- `$dotted-length`
- `$dotted-last`
- `$dotted-butlast`
- `$identifier?`
- `$free-identifier=?`
- `$syntax-rename`
- `$invalid-form`
- `$map-while`
- `$syntax-violation`
- `$generate-temporaries`
- `$make-variable-transformer`
- `$undefined`

6.3 Procedure Application

Syntax:

(*procedure* `arg-1` ...`arg-n`)

The procedure *procedure* is called with arguments `arg-1`, ..., `arg-n`. Note that it is legal to have an expression returning a simple procedure as the procedure to be called. It is an error if the type of any argument is `<none>`. When a procedure is called it is always checked that the types of the arguments are correct to that procedure. This check occurs either translation time or run time.

6.4 Instantiation of a Parametrized Type

Let A be a parametrized class or a parametrized logical type. Let a_1, ..., a_n be type expressions and t_1, ..., t_m be the translated argument list generated by them. Then the value of expression (A a_1 ...a_n) is an instance of parametrized type A with type parameter values t_1, ..., t_m. Two distinct instantiations of a parametrized class with same type parameter values shall refer to the same class.

6.5 Instantiation of Procedure Classes

Abstract and simple procedure classes are instantiated with the following syntax:

`(`*proc-metaclass argument-list result-type attribute-list* `)`

proc-metaclass ::=`:procedure` | `:simple-proc`
argument-list ::=`(`[arg_1 ...arg_n] `)`
attribute-list ::=`(`*attribute* ... `)`
attribute ::=`pure` | `nonpure`
| `always-returns` | `may-return`| `never-returns` | `static`

This syntax creates an abstract or simple procedure class. Expressions arg_1, ..., arg_n define the argument types. These expressions have to be static type expressions.

Parametrized procedure classes are instantiated with the following syntax:

`(:param-proc` *type-param-list argument-list result-type attribute-list* `)`

type-param-list ::=`(`[$tparam_1$ ...$tparam_m$] `)`
$tparam_k$ ::=identifier
argument-list ::=`(`[arg_1 ...arg_n] `)`
attribute-list ::=`(`*attribute* ... `)`
attribute ::=`pure` | `nonpure`
| `always-returns` | `may-return`| `never-returns` | `static`

Generic procedure classes cannot be instantiated explicitly for the moment.

6.6 Quotation

Quotation and quasiquotation work as in Scheme. Expression `(quote expr)` can be written `'expr` . Expression `(quasiquote expr)` can be written `‘expr` .

6.7 Implicit Declaration of Recursive Definitions

Keywords **define-simple-proc**, **define-param-proc**, **define-simple-method**, **define-param-method**, **define-class**, **define-param-class**, and **define-param-logical-type** declare the variables they define implicitly so that you do not have to declare them explicitly for recursion. However, mutually recursive definitions require declarations. Keywords **define-simple-proc**, **define-param-proc**, **define-simple-method**, and **define-param-method** are defined in the core library.

6.8 Module Forms

6.8.1 define-proper-program

Syntax:

(**define-proper-program** *program-name*
[*module-expression*] ...
[expression] ...)

program-name ::=module-name
module-expression ::=(module-keyword module-name ...)
module-name ::=identifier | (identifier ...)
module-keyword ::=**import** | **use** | **prelink-body**

A proper program with name *program-name* is defined. See chapter 3.

6.8.2 define-script

Syntax:

(**define-script** *program-name*
[*module-expression*] ...
[expression] ...)

program-name ::=module-name
module-expression ::=(module-keyword module-name ...)
module-name ::=identifier | (identifier ...)
module-keyword ::=**import** | **use** | **prelink-body**

A script with name *program-name* is defined. See chapter 3.

6.8.3 define-interface

Syntax:

(**define-interface** *mod-name*
[*module-expression*] ...
[*interface-expression*] ...)

mod-name ::=module-name
module-expression ::=(module-keyword module-name ...)
module-name ::=identifier | (identifier ...)
module-keyword ::=**import** | **import-and-reexport** | **use** | **prelink-body**
interface-expression ::=*declaration* | *definition*

An interface with name *mod-name* is defined. The *mod-name* may be either a single identifier or a list of identifiers. See chapter 3.

6.8.4 define-body

Syntax:

(**define-body** *mod-name*
[*module-expression*]
[expression] ...)

mod-name ::=module-name
module-expression ::=(module-keyword module-name ...)
module-name ::=identifier | (identifier ...)
module-keyword ::=**import** | **use** | **prelink-body**

A body with name *mod-name* is defined. The *mod-name* may be either a single identifier or a list of identifiers. See chapter 3.

6.8.5 import

Syntax:

(**import** module-name ...)

module-name ::=identifier | (identifier ...)

An interface is imported. See chapter 3 and subsections 6.8.1, 6.8.2, 6.8.3, and 6.8.4.

6.8.6 import-and-reexport

Syntax:

(**import-and-reexport** module-name ...)

module-name ::=identifier | (identifier ...)

An interface is imported and reexported. See chapter 3 and subsections 6.8.3, and 6.8.4.

6.8.7 use

Syntax:

(**use** module-name ...)

module-name ::=identifier | (identifier ...)

An interface can be accessed but its contents are not imported into the toplevel namespace. See chapter 3 and subsections 6.8.1, 6.8.2, 6.8.3, and 6.8.4.

6.8.8 @

Syntax:

(**@** module-name *variable*)

module-name ::=identifier | (identifier ...)
variable ::=identifier

Access a variable in the specified module. See chapter 3 and subsections 6.8.1, 6.8.2, 6.8.3, and 6.8.4.

6.8.9 reexport

Syntax:

(**reexport** identifier)

A variable is reexported. This expression type can occur only inside an interface. See chapter 3 and subsection 6.8.3.

6.8.10 prevent-stripping

Syntax:

(**prevent-stripping** identifier)

This expression prevents stripping off a procedure or a class from the linker output even though it is not detected in the coverage analysis. This should be necessary only with the foreign function interface.

6.8.11 prelink-body

Syntax:

(**prelink-body** module-name ...)

module-name ::=identifier | (identifier ...)

The bodies for the specified modules are linked before the unit where the **prelink-body** statement is given. Consequently the procedures defined in the prelinked bodies may be called toplevel in the unit. See chapter 3 and subsections 6.8.1, 6.8.2, 6.8.3, and 6.8.4.

6.9 Toplevel Definitions

6.9.1 define

Syntax:

(**define** *variable-name* [*type*] *value*)

variable-name ::=identifier

A constant with name *variable-name* and value *value* is defined. Expression *type* has to be a static type expression if it is present. If *type* is specified and *value* is not an instance of *type* an error is signalled.

6.9.2 define-class

Syntax:

(**define-class** *class-name superclass inheritable immutable eq-by-value ctr-access zero-value field-list*)

class-name ::=identifier
inheritable ::=boolean
immutable ::=boolean
eq-by-value ::=boolean
ctr-access ::=access-specifier
field-list ::=([*field-specifier*] ...)
field-specifier ::=(*field-name field-type read-access write-access* [*field-initial-value*])
field-name ::=identifier
read-access ::=access-specifier
write-access ::=access-specifier
access-specifier ::=`public` | `module` | `hidden`

A new class is defined. Parameter *superclass* has to be a static type expression whose value is a class. Parameters *field-type* have to be static type expressions.

6.9.3 define-generic-proc

Syntax:

(**define-generic-proc** *generic-name*)

generic-name ::=identifier

This expression defines a generic procedure with the name given. Note that **define-simple-method** and **define-param-method** define a generic procedure implicitly if it has not been already defined.

6.9.4 define-goops-class

Syntax:

(**define-goops-class** *name target-name superclass inheritable? immutable? equal-by-value? checked? zero-var equal-pred equal-contents-pred*)
name ::=identifier
target-name ::=identifier
inheritable? ::=boolean
immutable? ::=boolean
equal-by-value? ::=boolean
checked? ::=boolean
zero-var ::=identifier | null
equal-pred ::=identifier | null
equal-contents-pred ::=identifier | null

Keyword **define-goops-class** defines a custom GOOPS class existing in the target environment. A custom GOOPS class may only inherit (in Theme-D) from another custom GOOPS class or from `<object>`. Flags *inheritable?*, *immutable?* and *equal-by-value?* specify whether the class is inheritable, immutable or equal by value, respectively. If *checked?* is `#t` the types of the result values of the predicates are checked runtime. If *zero-var* is not null it defines a zero value (variable) for the class. Arguments *equal-pred* and *equal-contents-pred* determine the Scheme predicates that are used to compare values of this class in predicates `equal?` and `equal-contents?`. When procedure `equal-objects?` is used with GOOPS objects the comparison is performed by target Scheme procedure `eq?`.

6.9.5 define-mutable

Syntax:

(**define-mutable** *variable-name type value*)

variable-name ::=identifier

A mutable variable with name *variable-name*, type *type* and initial value *value* is defined. Expression *type* has to be a static type expression. If *value* is not an instance of *type* an error is signalled.

6.9.6 define-volatile

Syntax:

(**define-volatile** *variable-name type value*)

variable-name ::=identifier

A volatile variable with name *variable-name*, type *type* and initial value *value* is defined. Expression *type* has to be a static type expression. If *value* is not an instance of *type* an error is signalled.

6.9.7 define-param-logical-type

Syntax:

(**define-param-logical-type** *param-ltype-name type-parameter-list type-expression*)

param-ltype-name ::=identifier

Expression *type-expression* has to be a static type expression. When the instances of the parametrized logical type are created the type variables in *type-parameter-list* are bound to the values given for them and these bindings are applied for *type-expression*.

6.9.8 define-param-class

Syntax:

(**define-param-class** *param-class-name type-parameter-list superclass inheritable immutable eq-by-value ctr-access zero-value field-list*)

type-parameter-list ::=(*type-param*$_1$ ...*type-param*$_n$)
type-param$_k$ ::=identifier

The syntax of *type-parameter-list* is the same as in **define-param-logical-type**. The syntax of the last seven parameters is similar to their syntax in **define-class**. Parameter *superclass* has to be a static type expression whose value is a class. Parameters *field-type* have to be static type expressions. Parameters *superclass*, *inheritable*, *immutable*, *eq-by-value*, *ctr-access*, and the field list define the properties of the instances of the parametrized class being defined. When the instances of the parametrized class are created the type variables in *type-parameter-list* are bound to the values given for them and these bindings are applied for *field-list* and *superclass*.

6.9.9 define-param-proc-alt

Syntax:

(**define-param-proc-alt** *proc-name* ($type_1$ ... $type_n$) *proc-expression*)

proc-name ::=identifier
$type_k$ ::=identifier

This is an alternate way to define a parametrized procedure. Expression *proc-expression* has to be a **lambda** expression.

6.9.10 define-param-signature

Syntax:

(**define-param-signature** *signature-name type-param-list super proc-specifier* ...)

signature-name ::=identifier
type-param-list ::=([identifier ...])
super ::=identifier | null
proc-specifier ::=(*procedure-name arg-type-list result-type attribute-list*)
procedure-name ::=identifier
arg-type-list ::=([*arg-type* ...])
attribute-list ::=(*attribute* ...)
attribute ::=`pure` | `nonpure`
| `always-returns` | `may-return`| `never-returns` | `static`

Object *super* is the signature from which the parametrized signature inherits. In case a signature does not inherit anything *super* is set to null. A complete specifier list of a parametrized signature is obtained by concatenating the complete specifier list of the *super* signature with the specifier list of the parametrized signature being defined.

Expressions *proc-specifier* specify the procedures that all instances of the parametrized signature have to implement. Keyword **this** is used to refer to an instance of the parametrized signature itself in the procedure specifiers.

When the type variables of a parametrized signature are substituted with types we get an instance of the parametrized signature. This instance is a (ordinary) signature.

6.9.11 define-prim-class

Syntax:

(**define-prim-class** *name immutable? equal-by-value? checked? zero-var member-pred equal-pred equal-objects-pred equal-contents-pred*)
name ::=identifier
immutable? ::=boolean
equal-by-value? ::=boolean
checked? ::=boolean
zero-var ::=identifier | null
member-pred ::=identifier | null
equal-pred ::=identifier | null
equal-objects-pred ::=identifier | null
equal-contents-pred ::=identifier | null

Keyword **define-prim-class** defines a custom primitive class existing in the target environment. A custom primitive class cannot be inherited and it is an immediate descendant of `<object>`. Procedure *member-pred* determines if an object belongs to the class. Flags *immutable?* and *equal-by-value?* specify whether the class is immutable or equal by value, respectively. If *checked?* is `#t` the types of the result values of the predicates are checked runtime. If *zero-var* is not null it defines a zero value (variable) for the class. Arguments *equal-pred*, *equal-objects-pred*, and *equal-contents-pred* determine the target Scheme procedures that are used to compare the values of this class in predicates `equal?`, `equal-objects?`, and `equal-contents?`. If any of these arguments is null the default value `eqv?` is used.

6.9.12 define-signature

Syntax:

(**define-signature** *signature-name super proc-specifier ...*)

signature-name ::=identifier *super* ::=identifier | null
proc-specifier ::=(*procedure-name arg-type-list result-type attribute-list*)
procedure-name ::=identifier
arg-type-list ::=([*arg-type* ...])
attribute-list ::=(*attribute* ...)
attribute ::=`pure` | `nonpure`
| `always-returns` | `may-return`| `never-returns` | `static`

Object *super* is the signature from which the signature inherits. In case a signature does not inherit anything *super* is set to null. A complete specifier list of a signature is obtained by concatenating the complete specifier list of the *super* signature with the specifier list of the signature being defined.

Expressions *proc-specifier* specify the procedures that all instances of the signature have to implement. Keyword **this** is used to refer to the signature itself in the procedure specifiers.

6.9.13 add-method

Syntax:

(**add-method** *generic-name method*)

generic-name ::=identifier

Keyword **define-simple-method** adds *method* into the generic procedure *generic-name*. Procedure *method* has to be a simple procedure or a parametrized procedure.

6.10 Declarations

6.10.1 declare

Syntax:

(**declare** *variable-name class*)

variable-name ::=identifier

A **declare** expression declares a variable with given class without defining it. It is possible to use the variable after declaration although the use may be restricted somehow. E.g. it is not possible to use a declared class before defining it as a superclass of another class. Note that **declare** needs always a class and it does not accept logical types. Expression *class* has to be a static type expression whose value is a class. It is possible to redeclare the variable several times but then the new declared class has to be a subclass of the old class and the new class must have the same number of fields as the old class. The same typing rule is applied also when a declared variable is defined (the defined type is the new class). A declared variable has to be defined in the same module where the declaration is.

6.10.2 declare-method

Syntax:

(**declare-method** *generic-name procedure-class*)
generic-name ::=identifier

Keyword **declare-method** declares a method. A declaration of the method is added into the generic procedure *generic-name*. The *procedure-class* has to be either a simple or a parametrized procedure class. A declared method has to be defined either

- in the same translation unit where the declaration is or
- in the body of the interface if the declaration is in an interface.

6.10.3 declare-mutable

Syntax:

(**declare-mutable** *variable-name class*)

variable-name ::=identifier

Keyword **declare-mutable** declare a mutable variable. The *class* has to be the exact class of variable *variable-name*. Note that a variable declared with **declare-mutable** cannot be defined as volatile.

6.10.4 declare-volatile

Syntax:

(**declare-volatile** *variable-name class*)

variable-name ::=identifier

Keyword **declare-volatile** declares a volatile variable. The *class* has to be the exact class of variable *variable-name*.

6.11 Control Structures

6.11.1 if

Syntax:

(**if** *condition then-expression* [*else-expression*])

The type of *condition* has to be `<boolean>`. If *else-expression* is defined the type of the **if** expression is the union of the types of *then-expression* and *else-expression*. Otherwise the type of the **if** expression is `<none>`.

If *condition* is `#t` *then-expression* is evaluated. If *condition* is `#f` and *else-expression* is defined *else-expression* is evaluated. If the result type of the **if** expression is not `<none>` the value returned from *then-expression* or *else-expression* is returned from the **if** expression. Note that *then-expression* or *else-expression* are not necessarily evaluated at all.

6.11.2 if-object

Syntax:

(**if-object** *condition then-expression* [*else-expression*])

The *condition* can be any object. If *else-expression* is defined the type of the **if-object** expression is the union of the types of *then-expression* and *else-expression*. Otherwise the type of the **if-object** expression is `<none>`.

If *condition* is not `#f` *then-expression* is evaluated. If *condition* is `#f` and *else-expression* is defined *else-expression* is evaluated. If the result type of the **if-object** expression is not `<none>` the value returned from *then-expression* or *else-expression* is returned from the **if-object** expression. Note that *then-expression* or *else-expression* are not necessarily evaluated at all.

6.11.3 until

Syntax:

(**until** (*condition* [*result-expression*]) *body-expression*$_1$ ...*body-expression*$_n$)

The type of *condition* has to be `<boolean>`. At the beginning of each iteration *condition* is evaluated. If it returns `#t` the iteration is stopped and the value of *result-expression* is returned as the result of the **until** expression. Otherwise the body expressions are evaluated in order and the next iteration is started from the beginning. If *result-type* is not specified the type of the **until** expression is `<none>`.

6.11.4 begin

Syntax:

(**begin** *expr*$_1$ ...*expr*$_n$)

The type of the **begin** expression is the type of the last component expression *expr*$_n$. All the component expressions *expr*$_k$ are evaluated in order. If the result type of the last component expression is not `<none>` its value is returned as the value of the **begin** expression.

6.11.5 set!

Syntax:

(**set!** *variable-name value*)

variable-name ::=identifier

The value of the variable *variable-name* is set to *value*. Variable *variable-name* has to be defined and it has to be mutable. The type of *value* has to be a subtype of the type of variable *variable-name*. If these rules are violated a translation error (usually a compilation error) is signalled.

6.11.6 execute-with-current-continuation (exec/cc)

`(`**execute-with-current-continuation** | **exec/cc** *jump-proc jump-type body* `)`

jump-proc ::=identifier

This is a frontend for the built-in procedures

- `call-with-current-continuation`
- `call-with-current-continuation-nonpure`
- `call-with-current-continuation-without-result`

See sections 7.2.3, 7.2.4, and 7.2.5. The *body* is an expression that may invoke procedure *jump-proc*. This kind of invocation sets the current continuation to the continuation of the **exec/cc** expression. Type *jump-type* is the type of the object that *jump-proc* passes into the continuation. Keyword **exec/cc** is defined as an alias to **execute-with-current-continuation**.

6.11.7 generic-proc-dispatch

Syntax:

`(`**generic-proc-dispatch** *gen-proc-name* ($arg\text{-}type_1$ ... $arg\text{-}type_n$)
attribute-list `)`

gen-proc-name ::=identifier
attribute-list ::=`(`*attribute* ... `)`
attribute ::=`pure` | `nonpure`
| `always-returns` | `may-return`| `never-returns` | `static`

Keyword **generic-proc-dispatch** returns a simple procedure that dispatches a call to generic procedure *gen-proc-name* with argument type $arg\text{-}type_k$. Expressions $arg\text{-}type_k$ have to be static type expressions. The dispatched method must be compatible with the given attributes and its result type must not be `<none>`. Although a value of a **generic-proc-dispatch** expression is a simple procedure the dispatch is generally done runtime. Calling a **generic-proc-dispatch** expression always finds the correct method based on the methods contained in the generic procedure run time.

6.11.8 generic-proc-dispatch-without-result

Syntax:

(**generic-proc-dispatch-without-result** *gen-proc-name* (*arg-type*$_1$ ... *arg-type*$_n$) *attribute-list*)

gen-proc-name ::=identifier
attribute-list ::=(*attribute* ...)
attribute ::=`pure` | `nonpure`
| `always-returns` | `may-return`| `never-returns` | `static`

Keyword **generic-proc-dispatch-without-result** returns a simple procedure that dispatches a call to generic procedure *gen-proc-name* with argument type *arg-type*$_k$. Expressions *arg-type*$_k$ have to be static type expressions. The result type of the type of the dispatch expression is `<none>`. The dispatched method must be compatible with the given attributes. Although a value of a **generic-proc-dispatch-without-result** expression is a simple procedure the dispatch is generally done runtime. Calling a **generic-proc-dispatch-without-result** expression always finds the correct method based on the methods contained in the generic procedure run time.

6.11.9 param-proc-dispatch

Syntax:

(**param-proc-dispatch** *param-proc-name* *arg-type*$_1$ ...*arg-type*$_n$)

param-proc-name ::=identifier

A **param-proc-dispatch** expression returns a simple procedure obtained by creating an instance of parametrized procedure *param-proc-name*. The values of the type parameters of parametrized procedure *param-proc-name* are deduced from the types *arg-type*$_k$ as if the types *arg-type*$_k$ were argument types in an application of *param-proc-name*. Expressions *arg-type*$_k$ have to be static type expressions.

6.11.10 param-proc-instance

Syntax:

(**param-proc-instance** *param-proc-name* *arg-type*$_1$ ...*arg-type*$_n$)

param-proc-name ::=identifier

A **param-proc-instance** expression returns a simple procedure obtained by creating an instance of parametrized procedure *param-proc-name*. The type parameters defined in the definition of *param-proc-name* are bound to expressions $arg\text{-}type_k$ in order. Expressions $arg\text{-}type_k$ have to be static type expressions.

6.11.11 strong-assert

Syntax:

(**strong-assert** *condition*)

An assertion checks if the condition is true. If the condition is not true an exception will be raised. See also subsection 6.11.12. The difference between **assert** and **strong-assert** is that a strong assertion may never be neglected because of optimization.

6.11.12 assert

Syntax:

(**assert** *condition*)

An assertion checks if the condition is true. If the condition is not true an exception will be raised. See also subsection 6.11.11.

6.12 Macro Forms

6.12.1 define-syntax

Syntax:

(**define-syntax** *macro-name* *macro-transformer*)
macro-name ::=identifier

This form defines a macro.

6.12.2 let-syntax

Syntax:

(**let-syntax** ($var\text{-}spec_1$ ...$var\text{-}spec_n$) *let-syntax-body-expressions*)

$var\text{-}spec_k$ ::=($var\text{-}name_k$ $value_k$)

This form defines local macros.

6.12.3 letrec-syntax

Syntax:

(**letrec-syntax** (*var-spec*$_1$ ...*var-spec*$_n$) *let-syntax-body-expressions*)

var-spec$_k$ ::=(*var-name*$_k$ *value*$_k$)

This form defines local macros.

6.12.4 syntax-case

Syntax:

(**syntax-case** *expression* ([*literal*] ...)[*clause*] ...)
literal ::=identifier

This form defines a macro transformer.

6.13 Binding Forms

6.13.1 let

Syntax:

(**let** (*var-spec*$_1$ ... *var-spec*$_n$) *let-body-expressions*)

var-spec$_k$ ::=(*var-name*$_k$ [*var-type*$_k$] *value*$_k$)
var-name$_k$ ::=identifier
let-body-expressions ::=expression ...

Expressions *var-type*$_k$ have to be static type expressions. The result type of the let expression is the type of the last body expression. If the result type is not `<none>` the result value of the let expression is the value of the last body expression. The semantics of **let** expression is similar to these expressions in Scheme except the variable types are checked.

6.13.2 letrec and letrec*

Syntax:

({**letrec** | **letrec*** } (*var-spec*$_1$ ...*var-spec*$_n$) *letrec-body-expressions*)

var-spec$_k$ ::=`(`*var-name*$_k$ *var-type*$_k$ *value*$_k$ `)`
var-name$_k$ ::=identifier
letrec-body-expressions ::=expression ...

Expressions *var-type*$_k$ have to be static type expressions. The result type of the letrec expression is the type of the last body expression. If the result type is not `<none>` the result value of the letrec expression is the value of the last body expression. It is possible to refer to the letrec variables *var-name*$_k$ recursively in the expressions *value*$_k$ but these recursive uses of the variables must occur inside a **lambda** expression. Keyword **letrec*** differs from **letrec** so that **letrec*** guarantees to evaluate the expressions *value*$_k$ in order.

6.13.3 let-mutable, letrec-mutable, and letrec*-mutable

Syntax:

`(`{**let-mutable** | **letrec-mutable** | **letrec*-mutable** } `(`*var-spec*$_1$ ...*var-spec*$_n$ `)` *let-body-expressions* `)`

var-spec$_k$ ::=`(`*var-name*$_k$ *var-type*$_k$ *value*$_k$)
var-name$_k$ ::=identifier
let-body-expressions ::=expression ...

These expressions differ from the corresponding constant versions **let**, **letrec**, and **letrec*** so the variables *var-name*$_k$ are mutable in the **letxxx-mutable** expressions. Note that the variable types are compulsory in all of the **letxxx-mutable** expressions.

6.13.4 let-volatile, letrec-volatile, and letrec*-volatile

Syntax:

`(`{**let-volatile** | **letrec-volatile** | **letrec*-volatile** } `(`*var-spec*$_1$ ...*var-spec*$_n$ `)` *let-body-expressions* `)`

var-spec$_k$ ::=`(`*var-name*$_k$ *var-type*$_k$ *value*$_k$)
var-name$_k$ ::=identifier
let-body-expressions ::=expression ...

These expressions differ from the corresponding mutable versions **letxxx-mutable** so the variables *var-name*$_k$ are volatile in the **letxxx-volatile** expressions. Note that the variable types are compulsory in all of the **letxxx-volatile** expressions.

6.14 Procedure Creation

6.14.1 lambda

Syntax:

(**lambda** [*name*] (*argument-list result-type attribute-list*) *body-expr*$_1$, ..., *body-expr*$_n$)

name ::=identifier
argument-list ::=([*arg*$_1$ ...*arg*$_n$])
arg$_k$ ::=(*arg-name*$_k$ *arg-type*$_k$)
arg-name$_k$ ::=identifier
attribute-list ::=(*attribute* ...)| *attribute*
attribute ::=`pure` | `nonpure` | `force-pure`
| `always-returns` | `may-return`| `never-returns` | `static`

A **lambda** expression creates a simple procedure. Note that the argument list may be `()`. Expressions *arg-type*$_k$ and *result-type* have to be static type expressions. It is an error if the result type is not `<none>` and the type of the last body expression is not a subtype of *result-type*. If *result-type* is not `<none>` the result value of the procedure is the value of the last body expression. Expression *name* is the optional name of the lambda expression.

6.14.2 lambda-automatic

Syntax:

(**lambda-automatic** (*argument-list attribute-list*) *body-expr*$_1$, ..., *body-expr*$_n$)

argument-list ::=([*arg*$_1$ ...*arg*$_n$])
arg$_k$ ::=(*arg-name*$_k$ *arg-type*$_k$)
arg-name$_k$ ::=identifier
attribute-list ::=(*attribute* ...)| *attribute*
attribute ::=`pure` | `nonpure` | `force-pure`
| `always-returns` | `may-return`| `never-returns` | `static`

This form works as **lambda** except the result type is set to the type of the last body expression.

6.14.3 param-lambda

Syntax:

(**param-lambda** (*type*$_1$ ... *type*$_n$)(*argument-list result-type attribute-list*) *body-expr*$_1$, ..., *body-expr*$_n$)

$type_k$ ::=identifier
argument-list ::=`(`[*arg_1* ...*arg_n*] `)`
arg_k ::=(*$arg\text{-}name_k$* *$arg\text{-}type_k$*)
$arg\text{-}name_k$ ::=identifier
attribute-list ::=`(`*attribute* ... `)`| *attribute*
attribute ::=`pure` | `nonpure` | `force-pure`
| `always-returns` | `may-return`| `never-returns` | `static`

A **param-lambda** expression creates a parametrized procedure. Note that the argument list may be `()`. Expressions *$arg\text{-}type_k$* and *result-type* have to be static type expressions. It is an error if the result type is not `<none>` and the type of the last body expression is not a subtype of *result-type*. If *result-type* is not `<none>` the result value of the procedure is the value of the last body expression.

6.14.4 param-lambda-automatic

Syntax:

`(`**param-lambda-automatic** `(`*$type_1$* ... *$type_n$* `)(`*argument-list attribute-list* `)` *$body\text{-}expr_1$*, ..., *$body\text{-}expr_n$* `)`

$type_k$ ::=identifier
argument-list ::=`(`[*arg_1* ...*arg_n*] `)`
arg_k ::=(*$arg\text{-}name_k$* *$arg\text{-}type_k$*)
$arg\text{-}name_k$ ::=identifier
attribute-list ::=`(`*attribute* ... `)`| *attribute*
attribute ::=`pure` | `nonpure` | `force-pure`
| `always-returns` | `may-return`| `never-returns` | `static`

This form works as **param-lambda** except the result type is set to the type of the last body expression.

6.14.5 prim-proc and unchecked-prim-proc

Syntax:

`(`{**prim-proc** | **unchecked-prim-proc** }*procedure-name argument-list result-type attribute-list* `)`

procedure-name ::=identifier
argument-list ::=`(`[*$arg\text{-}type_1$* ...*$arg\text{-}type_n$*] `)`
attribute-list ::=`(`*attribute* ... `)`| *attribute*
attribute ::=`pure` | `nonpure` | `force-pure`
| `always-returns` | `may-return`| `never-returns` | `static`

With the current Theme-D implementation the target platform is Scheme (guile 2.0) and **prim-proc** defines a Theme-D procedure that calls a Scheme procedure *procedure-name*. Expressions *arg-type*$_k$ have to be static type expressions. If *result-type* is not `<none>` the Theme-D procedure also checks that the value returned from the Scheme procedure is an instance of *result-type*. The semantics of **unchecked-prim-proc** is similar to **prim-proc** except **unchecked-prim-proc** generates no run-time type checks for the result value.

6.14.6 param-prim-proc and unchecked-param-prim-proc

Syntax:

`(`{**param-prim-proc** | **unchecked-param-prim-proc** }*procedure-name type-parameter-list argument-list result-type attribute-list* `)`

procedure-name ::=identifier
type-parameter-list ::=`(`[*param*$_1$ ...*param*$_m$] `)`
param$_k$ ::=identifier
argument-list ::=`(`[*arg-type*$_1$ ...*arg-type*$_n$] `)`
attribute-list ::=`(`*attribute* ... `)`| *attribute*
attribute ::=`pure` | `nonpure` | `force-pure`
| `always-returns` | `may-return`| `never-returns` | `static`

With the current Theme-D implementation the target platform is Scheme (guile 2.0) and **param-prim-proc** defines a Theme-D parametrized procedure that calls a Scheme procedure *procedure-name*. Expressions *arg-type*$_k$ have to be static type expressions. If *result-type* is not `<none>` the Theme-D procedure also checks that the value returned from the Scheme procedure is an instance of *result-type*. The semantics of **unchecked-param-prim-proc** is similar to **param-prim-proc** except **unchecked-prim-proc** generates no run-time type checks for the result value.

6.15 Type Operations

6.15.1 cast

Syntax:

`(`**cast** *type casted-value* `)`

The static type of the **cast** expression is the value of *type*. Expression *type* has to be a static type expression. It is an error (translation time or run-time) if the result value of *casted-value* is not an instance of *type*. See also subsection 6.15.2.

6.15.2 try-cast

Syntax:

(**try-cast** *type* *casted-value* *default-value*)

If *casted-value* is an instance of *type* return *casted-value*. Otherwise return *default-value*. The static type of the **try-cast** expression is the union of *type* and the type of *default-value*. Expression *type* has to be a static type expression. See also subsection 6.15.1.

6.15.3 force-pure-expr

Syntax:

(**force-pure-expr** *expr*)

This form makes the component expression *expr* to appear pure for Theme-D. See section 8.2 for example usage of the form.

6.15.4 match-type

Syntax:

(**match-type** *value-to-match* [*clause-list*] [*else-clause*])

clause-list ::=$clause_1$, ..., $clause_n$
$clause_k$ ::=($match\text{-}spec_k$ $expr_{k,1}$, ..., $expr_{k,m(k)}$)
$match\text{-}spec_k$ ::=(var_k $type_k$)| ($type_k$)
else-clause ::=(**else** $else\text{-}expr_1$, ..., $else\text{-}expr_p$)

Each clause is processed in order. If var_k is given and the runtime type of *value-to-match* is a subtype of $type_k$ then bind var_k to *value-to-match*, evaluate expressions $expr_{k,1}$, ..., $expr_{k,m(k)}$ is order and return the value of the last expression. The static type of var_k is $type_k$. If var_k is not given and the runtime type of *value-to-match* is a subtype of $type_k$ then evaluate expressions $expr_{k,1}$, ..., $expr_{k,m(k)}$ is order and return the value of the last expression. If none of the types matches and *else-clause* is present evaluate the expressions $else\text{-}expr_1$, ..., $else\text{-}expr_p$ and return the value of the last expression.

Let K be an integer between 1 and n and u the union of types $type_k$, $k = 1, \ldots, K$. If the static type of *value-to-match* is a subtype of u the following optimizations are done:

- The clause $clause_K$ needs no runtime type check because we already known that the type of *value-to-match* is a subtype of $type_K$.

- If all the type checks $k = 1, \ldots, K-1$ fail the expressions of clause K are automatically evaluated. Consequently the clauses $k > K$ and the else clause need not be compiled.

6.15.5 static-type-of

Syntax:

(**static-type-of** *expression*)

This form returns the static type of *expression*. This computation is done compile time and *expression* is not evaluated run time.

6.15.6 :tuple

Syntax:

(**:tuple** a_1 ...a_n)

Let u ::=(t_1 ...t_m) be the translated argument list generated from a_1 ...a_n . Object of type u is a list with element types t_1 ...t_m . Expression (**:tuple** a_1 ...a_n) is equivalent to `(:pair` t_1 `(:pair` t_2 (...`(:pair` t_n `<null>`)...)))`.

6.16 Object Creation

6.16.1 constructor

Syntax:

(**constructor** *class*)

Expression *class* has to be a static type expression and its value has to be a class. The value of a **constructor** expression is the constructor (a simple procedure) of class *class*.

6.16.2 quote

Syntax:

(**quote** *quoted-expression*)

Keyword **quote** is used to create atom and list constants as in Scheme.

6.16.3 zero

Syntax:

(**zero** *class*)

Keyword **zero** accesses the zero value of a class. It is an error to use **zero** for a class that does not define a zero value.

Chapter 7

Special Procedures

Special procedures are procedures that are treated specially be Theme-D compiler and linker. They are typically parametrized procedures whose typing cannot be expressed in current Theme-D. Note that the types we give for the arguments of the special procedures do not generally describe all the requirements the special procedures have for argument types. The application procedures `apply` and `apply-nonpure`, could be implemented in Theme-D but they have been included in the core language because of optimization.

7.1 Equality Predicates

7.1.1 `equal?`

Syntax:

```
(equal?  object1 object2)
```

Arguments:

Name: `object1`
Type: `<object>`
Description: An object to be compared

Name: `object2`
Type: `<object>`
Description: An object to be compared

Result value: `#t` iff `object1` is equal to `object2`
Result type: `<boolean>`

Purity of the procedure: pure

This is the main equality predicate in Theme-D. This procedure implements algorithm Equal?, see section 4.14.2. Name `=` is an alias for `equal?`.

7.1.2 `equal-objects?`

Syntax:

```
(equal-objects? object1 object2)
```

Arguments:

Name: `object1`
Type: `<object>`
Description: An object to be compared

Name: `object2`
Type: `<object>`
Description: An object to be compared

Result value: `#t` iff `object1` is the same object as `object2`
Result type: `<boolean>`

Purity of the procedure: pure

This procedure implements algorithm EqualObjects?, see section 4.14.4.

7.1.3 `equal-contents?`

Syntax:

```
(equal-contents? object1 object2)
```

Arguments:

Name: `object1`
Type: `<object>`
Description: An object to be compared

Name: `object2`
Type: `<object>`
Description: An object to be compared

Result value: `#t` iff the contents of `object1` are equal to the contents of `object2`
Result type: `<boolean>`

Purity of the procedure: pure

This procedure implements algorithm EqualContents?, see section 4.14.3.

7.2 Control Structures

7.2.1 `apply`

Syntax:

```
(apply procedure argument-list)
```

Type parameters: `%arglist, %result`

Arguments:

Name: `procedure`
Type: `(:procedure ((`**splice** `%arglist)) %result pure)`
Description: procedure to be called

Name: `argument-list`
Type: `%arglist`
Description: arguments to be passed

Result value: The value returned from `procedure`
Result type: The result type of `procedure`

Purity of the procedure: pure

The type of `argument-list` has to be a subtype of the argument list type of `procedure`. Procedure `procedure` has to be pure. Procedure `apply` calls `procedure` with the arguments from `argument-list`. This is similar to Scheme `apply`.

7.2.2 `apply-nonpure`

Syntax:

```
(apply-nonpure procedure argument-list)
```

Type parameters: `%arglist, %result`

Arguments:

Name: `procedure`
Type: `(:procedure ((`**splice** `%arglist)) %result nonpure)`
Description: procedure to be called

Name: `argument-list`
Type: `%arglist`
Description: arguments to be passed

Result value: The value returned from `procedure`
Result type: The result type of `procedure`

Purity of the procedure: nonpure

The type of `argument-list` has to be a subtype of the argument list type of `procedure`. Procedure `procedure` may be pure or nonpure. Procedure `apply-nonpure` calls `procedure` with the arguments from `argument-list`. This is similar to Scheme `apply`.

7.2.3 `call-with-current-continuation` (`call/cc`)

Syntax:

```
(call-with-current-continuation body )
```

Type parameters: `%body-type, %jump-type`

Arguments:

Name: `body`
Type: `(:procedure ((:procedure (%jump-type) <none> pure)) %body-type pure)`
Description: The procedure to be called

Result value: Either the value of the body procedure or a value passed into the jump procedure
Result type: `(:union %body-type %jump-type)`

Purity of the procedure: pure

This procedures is a built-in parametrized procedure that works as the corresponding procedure in Scheme. The argument `body` is a procedure taking a single procedure argument. If the body procedure invokes this argument the continuation is transferred into the continuation of the `call-with-current-continuation` expression. Variable `call/cc` is defined as an alias to `call-with-current-continuation`.

7.2.4 `call-with-current-continuation-nonpure` (`call/cc-nonpure`)

Syntax:

```
(call-with-current-continuation-nonpure body )
```

Type parameters: `%body-type, %jump-type`

Arguments:

Name: `body`

Type: `(:procedure ((:procedure (%jump-type) <none> pure)) %body-type nonpure)`
Description: The procedure to be called

Result value: Either the value of the body procedure or a value passed into the jump procedure
Result type: `(:union %body-type %jump-type)`

Purity of the procedure: nonpure

This is a nonpure version of `call-with-current-continuation`, see the previous section. This procedure has an alias `call/cc-nonpure`.

7.2.5 `call-with-current-continuation-without-result` (`call/cc-without-result`)

Syntax:

```
(call-with-current-continuation-without-result body )
```

Arguments:

Name: `body`
Type: `(:procedure ((:procedure () <none> pure)) <none> nonpure)`
Description: The procedure to be called

No result value.

Purity of the procedure: nonpure

This is a version of `call-with-current-continuation` having no result value. This procedure has an alias `call/cc-without-result`.

7.2.6 `field-ref`

Syntax:

```
(field-ref object field-name)
```

Arguments:

Name: `object`
Type: `<object>`
Description: object whose field is accessed

Name: `field-name`
Type: `<symbol>`
Description: name of the field to be accessed

Result value: Value of the field
Result type: Type of the field

Purity of the procedure: pure

Argument `field-name` has to be a literal symbol.

7.2.7 `field-set!`

Syntax:

```
(field-ref object field-name field-value)
```

Arguments:

Name: `object`
Type: `<object>`
Description: object whose field is to be set

Name: `field-name`
Type: `<symbol>`
Description: name of the field to be set

Name: `field-value`
Type: `<object>`
Description: new value of the field

Result value: No result value
Result type: `<none>`

Purity of the procedure: nonpure

Argument `field-name` has to be a literal symbol. Argument `field-value` has to be an instance of the type of the field.

7.3 Type Operations

7.3.1 `class-of`

Syntax:

```
(class-of object)
```

Arguments:

Name: `object`
Type: `<object>`

Description: the object whose class is accessed

Result value: Class of the object
Result type: `<class>`

Purity of the procedure: pure

7.3.2 `is-instance?`

Syntax:

`(is-instance? object` *type*`)`

Arguments:

Name: `object`
Type: `<object>`
Description: An object whose type is checked

Name: *type*
Type: `<type>`
Description: A type

Result value: `#t` iff `object` is an instance of *type*
Result type: `<boolean>`

Purity of the procedure: pure

Argument *type* has to be a static type expression. Expression

`(is-instance? object` *type*`)`

is equivalent to

`(is-subtype? (class-of object)` *type*`)`

7.3.3 `is-subtype?`

Syntax:

`(is-subtype?` $type_1$ $type_2$`)`

Arguments:

Name: $type_1$
Type: `<type>`

Description: A type

Name: *type*$_2$
Type: `<type>`
Description: A type

Result value: `#t` iff *type*$_1$ is a subtype of *type*$_2$
Result type: `<boolean>`

Purity of the procedure: pure

Arguments *type*$_1$ and *type*$_2$ have to be a static type expressions.

7.4 Vector Operations

7.4.1 `cast-mutable-value-vector`

Syntax:

(cast-mutable-value-vector *target-element-type* source-vector)

Arguments:

Name: *target-element-type*
Type: `<type>`
Description: Element type of the new vector

Name: `source-vector`
Type: any vector class
Description: The vector to be casted

Result value: A copy of the source vector with the new element type
Result type: (:mutable-value-vector *target-element-type*)

Purity of the procedure: pure

Special procedure `cast-mutable-value-vector` creates a copy of the source vector and checks that each of its elements is an instance of *target-element-type*. The check is generally done run time. The vector metaclass may change in the cast.

7.4.2 `cast-mutable-value-vector-metaclass`

Syntax:

```
(cast-mutable-value-vector-metaclass source-vector)
```

Arguments:

Name: `source-vector`
Type: any vector class
Description: The vector to be casted

Result value: A copy of the source vector with the new metaclass
Result type: `(:mutable-value-vector` *element-type*`)`

Purity of the procedure: pure

Special procedure `cast-mutable-value-vector-metaclass` creates a copy of the source vector so that the copy has the class `(:mutable-value-vector` *element-type*`)` where *element-type* is the element type of the original vector.

7.4.3 `cast-mutable-vector`

Syntax:

(`cast-mutable-vector` *target-element-type* `source-vector)`

Arguments:

Name: *target-element-type*
Type: `<type>`
Description: Element type of the new vector

Name: `source-vector`
Type: any vector class
Description: The vector to be casted

Result value: A copy of the source vector with the new element type
Result type: `(:mutable-vector` *target-element-type*`)`

Purity of the procedure: pure

Special procedure `cast-mutable-vector` creates a copy of the source vector and checks that each of its elements is an instance of *target-element-type*. The check is generally done run time. The vector metaclass may change in the cast.

7.4.4 `cast-mutable-vector-metaclass`

Syntax:

```
(cast-mutable-vector-metaclass source-vector)
```

Arguments:

Name: `source-vector`
Type: any vector class

Description: The vector to be casted

Result value: A copy of the source vector with the new metaclass
Result type: (`:mutable-vector` *element-type*)

Purity of the procedure: pure

Special procedure `cast-mutable-vector-metaclass` creates a copy of the source vector so that the copy has the class (`:mutable-vector` *element-type*) where *element-type* is the element type of the original vector.

7.4.5 `cast-value-vector`

Syntax:

(`cast-value-vector` *target-element-type* `source-vector`)

Arguments:

Name: *target-element-type*
Type: `<type>`
Description: Element type of the new vector

Name: `source-vector`
Type: any vector class
Description: The vector to be casted

Result value: A copy of the source vector with the new element type
Result type: (`:value-vector` *target-element-type)*

Purity of the procedure: pure

Special procedure `cast-value-vector` creates a copy of the source vector and checks that each of its elements is an instance of *target-element-type*. The check is generally done run time. The vector metaclass may change in the cast.

7.4.6 `cast-value-vector-metaclass`

Syntax:

```
(cast-value-vector-metaclass source-vector)
```

Arguments:

Name: `source-vector`
Type: any vector class
Description: The vector to be casted

Result value: A copy of the source vector with the new metaclass
Result type: `(:value-vector` *element-type*`)`

Purity of the procedure: pure

Special procedure `cast-value-vector-metaclass` creates a copy of the source vector so that the copy has the class `(:value-vector` *element-type*`)` where *element-type* is the element type of the original vector.

7.4.7 `cast-vector`

Syntax:

```
(cast-vector target-element-type source-vector)
```

Arguments:

Name: *target-element-type*
Type: `<type>`
Description: Element type of the new vector

Name: `source-vector`
Type: any vector class
Description: The vector to be casted

Result value: A copy of the source vector with the new element type
Result type: `(:vector` *target-element-type*`)`

Purity of the procedure: pure

Special procedure `cast-vector` creates a copy of the source vector and checks that each of its elements is an instance of *target-element-type*. The check is generally done run time. The vector metaclass may change in the cast.

7.4.8 `cast-vector-metaclass`

Syntax:

```
(cast-vector-metaclass source-vector)
```

Arguments:

Name: `source-vector`
Type: any vector class
Description: The vector to be casted

Result value: A copy of the source vector with the new metaclass
Result type: `(:vector` *element-type*`)`

Purity of the procedure: pure

Special procedure `cast-vector-metaclass` creates a copy of the source vector so that the copy has the class `(:vector` *element-type*`)` where *element-type* is the element type of the original vector.

7.4.9 `make-mutable-value-vector`

Syntax:

`(make-mutable-value-vector` *element-type* `nr-of-elements element-value)`

Arguments:

Name: *element-type*
Type: `<type>`
Description: Type of the vector elements

Name: `nr-of-elements`
Type: `<integer>`
Description: Number of elements in the new vector

Name: `element-value`
Type: `<object>`
Description: Value with which the new vector is filled

Result value: A mutable value vector of `nr-of-elements` elements with value `element-value`
Result type: `(:mutable-value-vector` *element-type*`)`

Purity of the procedure: pure

Argument *element-type* has to be a static type expression. Argument `element-value` has to be an instance of *element-type*.

7.4.10 `make-mutable-vector`

Syntax:

`(make-mutable-vector` *element-type* `nr-of-elements element-value)`

Arguments:

Name: *element-type*
Type: `<type>`
Description: Type of the vector elements

Name: `nr-of-elements`

Type: `<integer>`
Description: Number of elements in the new vector

Name: `element-value`
Type: `<object>`
Description: Value with which the new vector is filled

Result value: A mutable vector of `nr-of-elements` elements with value `element-value`
Result type: `(:mutable-vector` *element-type*`)`

Purity of the procedure: pure

Argument *element-type* has to be a static type expression. Argument `element-value` has to be an instance of *element-type*.

7.4.11 `make-value-vector`

Syntax:

`(make-value-vector` *element-type* `nr-of-elements element-value)`

Arguments:

Name: *element-type*
Type: `<type>`
Description: Type of the vector elements

Name: `nr-of-elements`
Type: `<integer>`
Description: Number of elements in the new vector

Name: `element-value`
Type: `<object>`
Description: Value with which the new vector is filled

Result value: A value vector of `nr-of-elements` elements with value `element-value`
Result type: `(:value-vector` *element-type*`)`

Purity of the procedure: pure

Argument *element-type* has to be a static type expression. Argument `element-value` has to be an instance of *element-type*.

7.4.12 `make-vector`

Syntax:

`(make-vector` *element-type* `nr-of-elements element-value)`

Arguments:

Name: *element-type*
Type: `<type>`
Description: Type of the vector elements

Name: `nr-of-elements`
Type: `<integer>`
Description: Number of elements in the new vector

Name: `element-value`
Type: `<object>`
Description: Value with which the new vector is filled

Result value: A vector of `nr-of-elements` elements with value `element-value`
Result type: `(:vector` *element-type*`)`

Purity of the procedure: pure

Argument *element-type* has to be a static type expression. Argument `element-value` has to be an instance of *element-type*.

7.4.13 `mutable-value-vector`

Syntax:

`(mutable-value-vector` *element-type* `element-1 ... element-n)`

Arguments:

Name: *element-type*
Type: `<type>`
Description: The element type of the new vector

Name: `element-k`
Type: `<object>`
Description: An element of the new vector

Result value: A mutable value vector with element type *element-type* and elements `element-1`, ..., `element-n`
Result type: `(:mutable-value-vector` *element-type*`)`

Purity of the procedure: pure

Argument *element-type* has to be a static type expression. Each `element-k` has to be an instance of *element-type*.

7.4.14 `mutable-vector`

Syntax:

(`mutable-vector` *element-type* `element-1 ... element-n`)

Arguments:

Name: *element-type*
Type: `<type>`
Description: The element type of the new vector

Name: `element-k`
Type: `<object>`
Description: An element of the new vector

Result value: A mutable vector with element type `element-type` and elements `element-1`, ..., `element-n`
Result type: (`:mutable-vector` *element-type*)

Purity of the procedure: pure

Argument *element-type* has to be a static type expression. Each `element-k` has to be an instance of *element-type*.

7.4.15 `value-vector`

Syntax:

(`value-vector` *element-type* `element-1 ... element-n`)

Arguments:

Name: *element-type*
Type: `<type>`
Description: The element type of the new vector

Name: `element-k`
Type: `<object>`
Description: An element of the new vector

Result value: A value vector with element type `element-type` and elements `element-1`, ..., `element-n`
Result type: (`:value-vector` *element-type*)

Purity of the procedure: pure

Argument *element-type* has to be a static type expression. Each `element-k` has to be an instance of *element-type*.

7.4.16 `vector`

Syntax:

`(vector` *element-type* `element-1 ... element-n)`

Arguments:

Name: *element-type*
Type: `<type>`
Description: The element type of the new vector

Name: `element-k`
Type: `<object>`
Description: An element of the new vector

Result value: A vector with element type `element-type` and elements `element-1`, ..., `element-n`
Result type: `(:vector` *element-type*`)`

Purity of the procedure: pure

Argument *element-type* has to be a static type expression. Each `element-k` has to be an instance of *element-type*.

7.5 Tuple Operations

7.5.1 `tuple-ref`

Syntax:

`(tuple-ref tuple index)`

Arguments:

Name: `tuple`
Type: `(:tuple`t_1 `...` t_n`)`
Description: A tuple

Name: `index`
Type: `<integer>`
Description: Index of the element wanted

Result value: The element of `tuple` at position `index`
Result type: $t_{\texttt{index}}$

The indices of a tuple start from zero.

7.5.2 `tuple-type-with-tail`

Syntax:

(`tuple-type-with-tail` *tuple-t* *tail-t*)

Arguments:

Name: *tuple-t*
Type: `<type>`
Description: A tuple type

Name: *tail-t*
Type: `<type>`
Description: A type

Result value: A list type constructed from *tuple-t* and *tail-t*
Result type: `<type>`

Let *tuple-t* be a tuple type consisting of types t_1, ..., t_n and *tail-t* be a type. Object of type (`tuple-type-with-tail` *tuple-t* *tail-t*) is a list with first n element types t_1, ..., t_n and the tail of the nth element *tail-t*. Expression (`tuple-type-with-tail` *tuple-t* *tail-t*) is equivalent to

(`:pair` t_1 (`:pair` t_2 (... (`:pair` t_n *tail-t*) ...)))

Chapter 8

Examples

Subdirectory `theme-d/theme-d-code/examples` contains some examples to illustrate the Theme-D programming language. Subdirectory `theme-d/theme-d-code/tests` contains programs and modules used to test the Theme-D system.

8.1 Abstract Data Types

Abstract data types are data types for which the data type is defined by specifying the operations (procedures) that the members of the data type have to implement. In Theme-D ADT's can be implemented either by using (parametrized) signatures or delegation.

We define the ADT's "sequence" and "association" as examples. The following operations are implemented by every sequence class:

- `sequence-length` that obtains the length of a sequence
- `sequence-ref` that obtains a sequence element at the given index
- `sequence-map` that maps a given procedure to a sequence

Associations resemble associations lists. The following operations are implemented by every association class:

- `gen-assoc` that obtains a value belonging to the given key
- `gen-assoc-set!` that adds a binding with given key and value into the association.

Files `sequence-sgn-test.thp`, `sequence-sgn.th?`, and `sequence-list-impl.th?` contain an implementation of ADT sequence implemented with parametrized signatures. Files `list-as-sequence.th?`, `vector-as-sequence.th?`, and `sequence-test.thp` contain an implementation of ADT sequence implemented by delegation. Files `assoc-test.thp`, `assoc-test2.thp`, `assoc-list-impl.th?`, `assoc-sgn.th?`, `hash-table.th?` and `singleton.th?` contain and implementation of ADT association implemented with parametrized signatures.

8.2 Creators (high-level constructors)

Sometimes the Theme-D constructors are not flexible enough to construct instances. It is possible to emulate GOOPS-style constructors in Theme-D. We give an example here. Example code can be found from

```
theme-d-code/examples/creators.thp
```

First define a general purpose generic procedure and macro for creating objects:

```
(define-generic-proc initialize)

(define-syntax create
  (syntax-rules ()
   ((create clas arg ...)
    (force-pure-expr
     (let ((tmp (make clas)))
       (initialize tmp arg ...)
      tmp)))))
```

Next define some classes to be used:

```
(define-class <graphics-context> <object> #t #f #f public ()
  ((i-x <integer> public public)
   (i-y <integer> public public)
   (i-width <integer> public public)
   (i-height <integer> public public)))

(define-class <widget> <object> #t #f #f public ()
  ((context (:maybe <graphics-context>) public module null)
   (widget-parent (:maybe <widget>) public module null)))

(define-class <label> <widget> #t #f #f public ()
  ((str-text <string> public module "")))
```

Define then the creator for the base class

```
(define-simple-method initialize
     (((widget <widget>) (context <graphics-context>)
       (widget-parent (:maybe <widget>)))
      <none> nonpure)
  (logger-print "initialize widget")
  (field-set!  widget 'context context)
  (field-set!  widget 'widget-parent widget-parent))
```

and for the derived class

```
(define-simple-method initialize
     (((label <label>) (context <graphics-context>)
      (widget-parent <widget>)
      (str-text <string>))
     <none> nonpure)
  (logger-print "initialize label")
  ((generic-proc-dispatch-without-result
    initialize
    (<widget> <graphics-context> <widget>)
    ())
   label context widget-parent)
  (field-set!  label 'str-text str-text))
```

Now the instances of the classes can be created as follows:

```
(let* ((widget-parent
        (create <widget>
          (make <graphics-context> 100 100 400 200)
          null))
       (label
        (create <label>
          (make <graphics-context> 100 100 200 50)
          widget-parent
          "Hello")))
 ...  )
```

Corresponding example for parametrized classes can be found from

```
theme-d-code/examples/param-creators.thp
```

In order to define creators follow the following guidelines:

- All the fields in the classes for which the creators are implemented shall define initial value. Note that you may have to declare types of some fields as `(:maybe <myclass>)` in order to allow `null` as the initial value.
- Define method `initialize` with first argument having the class for which the instances are created and possibly some other arguments.
- For each method `initialize` call the `initialize` method of the superclass by explicitly dispatching the method.
- When instances of the class are created its `initialize` method has to be called.
- If your creators have side effects other than setting the field values create separate macro `create-nonpure` and generic procedure `initialize-nonpure` for them:

```
(define-generic-proc initialize-nonpure)
```

```
(define-syntax create-nonpure
  (syntax-rules ()
   ((create clas arg ...)
    (let ((tmp (make clas)))
     (initialize-nonpure tmp arg ...)
     tmp)))))
```

Methods `initialize-nonpure` may call methods `initialize` in super classes.

8.3 Invoking the match-type Optimization

Consider the following code of a mapping functional:

```
(define-param-proc map1
     (%argtype %result-type)
     (((proc (:procedure (%argtype) %result-type pure))
      (lst (:uniform-list %argtype)))
     (:uniform-list %result-type)
     pure)
  (match-type lst
     ((<null>) null)
     ((lst1 (:nonempty-uniform-list %argtype))
      (cons (proc (car lst1))
            (map1 proc (cdr lst1))))))
```

Now the clause for `lst1` needs no runtime type check. We only need to check if `lst` is `null`.

8.4 Purely Functional Iterators

See [1] for discussion about purely functional iterators. In Theme-D purely functional iterators are implemented in module `(standard-library iterator)`. See program `theme-d-code/tests/test470.thp` for a demonstration about iterators.

Chapter 9

Comments

- Consider the test program `(tests test29)` and the application of procedure `apply` at the end of the procedure `my-map`. Procedure `apply` applies parametrized procedure `my-map` and the application is dispatched runtime. If the components of the argument list `cdrs` are `null` the type parameter `%arglist` in `my-map` cannot be deduced and the dispatch fails. Consequently we get a runtime error. A solution to this problem is to check that `cdrs` does not contain `null` values before the recursive application, see `(tests test30)`.
- The type correctness of the implementations of parametrized methods cannot always be checked translation time.
- Multiple inheritance is not going to be implemented in Theme-D. Single inheritance arises naturally from the memory layout of objects, i.e. a pointer to a derived class is also a pointer to the base class. This is not true in case of multiple inheritance.
- The procedures implementing the DeduceXXX algorithms can be found in file `theme-d-type-system.scm`, procedures `deduce-xxx`.
- The Theme-D runtime environment in

  ```
  theme-d/runtime/runtime-theme-d-environment.scm
  ```

  also contains procedures implementing the SelectBestMatch algorithm since the procedure dispatch is usually done run-time.

Bibliography

[1] H. G. Baker. Iterators: signs of weakness in object oriented languages. *ACM OOPS Messenger*, 4(3):18–25, 1993. http://www.pipeline.com/∼hbaker1/Iterator.html.

[2] H. Barendregt, W. Dekkers, and R. Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[3] M. S. et al. Revised^6 Report on the Algorithmic Language Scheme. 2007. http://www.r6rs.org/.

[4] J. Weel. Theme, a functional systems programming language. 2007. http://www.ugcs.caltech.edu/∼weel/theme.php.

Index