A Semantic Definition of Separate Type Checking in C++ with Concepts

Abstract syntax and the complete semantic definition

Marcin Zalewski

Technical Report No. 2008:12

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Göteborg, 2008

Contents

1 Introduction **6**

2 Syntax **6**

2.1 Grammar Rules . . . 7

id_τ type identifier . . . 7

$\overline{id_\tau}$ type identifier sequence . . . 7

con concept name . . . 7

$cid,\ cid_{cmap},\ cid_{impl}$ concept identifier . . . 7

$cid?$ concept identifier option . . . 7

$\overline{cid},\ \overline{cid}^{\prec_1}$ concept identifier sequence . . . 7

$scope$ scope (used in qualified names) . . . 7

req requirements clause . . . 8

τ type specifier . . . 8

$bintype$ built-in types . . . 8

$\tau?$ type specifier option . . . 8

$\overline{\tau}$ type specifier sequence . . . 8

val values . . . 8

e expressions . . . 8

Γ type environment $(var \rightharpoonup \tau)$ . . . 8

$tparam$ type parameter . . . 8

tl template . . . 9

T templates environment . . . 9

ty^a associated type . . . 9

$\overline{ty^a},\ \overline{ty^a}^{\prec}$ associated type sequence . . . 9

f^a associated function . . . 9

$\overline{f^a},\ \overline{f^a}^{\prec}$ associated function sequence . . . 9

cp concept . . . 9

$cp?$ concept option . . . 9

C concept sequence . . . 9

sig function signature . . . 9

$sig?$ function signature option . . . 9

$sig?_{amb}$ signature option or ambiguity . . . 10

$\overline{sig},\ \overline{sig}^{\prec}$ function signature sequence . . . 10

$fdef$ function definition . . . 10

$fdef?$ function declaration option . . . 10

$\overline{fdef},\ \overline{fdef}^{\prec}$ function definition sequence . . . 10

$tydef$ type definition . . . 10

$\overline{tydef},\ \overline{tydef}^{\prec_1}$ type definition sequence . . . 10

$tydef?$ type definition option . . . 10

cm concept map . . . 10

$\overline{cm}$ concept map sequence . . . 11

$cm?$ concept map option . . . 11

$\overline{cm?}$ concept map option sequence . . . 11

icm implicit concept map . . . 11

$\overline{icm}$ implicit concept map sequence . . . 11
$icm?$ implicit concept map option . . . 11
M concept map environment . . . 11
d_{user} user definitions . . . 11
P_{user} user syntax program . . . 11
d definitions . . . 11
P program . . . 12
terminals pretty printing of terminals . . . 12
formula formulas that can be used in premises . . . 14
fdecls_name_def . . . 15
fdecl_return_def . . . 15
fdecl_params_def . . . 15
construct_function_def . . . 15
cid_name_def . . . 15
concept_name_def . . . 15
concept_functions_def . . . 15
cid_functions_def . . . 15
cmap_functions_def . . . 15
concept_fields_def . . . 15
cid_types_def . . . 16
cmap_types_def . . . 16
cid_refined_cids_def . . . 16
con_refined_cids_def . . . 16
find_fcall_fdecls_def . . . 16
refines_dir_many . . . 16
cid_refines_many . . . 16
con_refines_many . . . 16
find_type_def . . . 16
find_type_rec_def . . . 16
find_fun_def . . . 16
get_overset_def . . . 16
normalize_fdecls_def . . . 16
find_fun_rec_def . . . 16
T_def . . . 17
typedecl_wf_def . . . 17
required_assfs_def . . . 17
required_assts_def . . . 17
defined_fdecls_ref_def . . . 17
defined_fdefs_ref_def . . . 17
defined_assts_def . . . 17
assts_compat_def . . . 17
typedecls_check_def . . . 17
typedecls_comatibility_def . . . 17

$\boxed{fdefs_check_def}$ . . . 17
$\boxed{fdefs_compat_def}$ . . . 17
$\boxed{implicit_cmaps_def}$ . . . 17
$\boxed{p_inst_def}$ . . . 17
$\boxed{p_user_inst_def}$ . . . 18
$\boxed{judgement}$ . . . 18
$\boxed{user_syntax}$ . . . 18
2.2 Subrules . . . 20

3 Helper Judgements **20**
3.1 $\boxed{fname\,(sig) = f}$ extract function name from a function signature . . . 20
3.2 $\boxed{returns\,(sig) = \tau}$ extract return type from a function signature . . . 20
3.3 $\boxed{params\,(sig) = \overline{\tau}}$ extract parameters from a function signature . . . 21
3.4 $\boxed{sig\,(f, \tau, \overline{\tau}) = sig}$ construct function signature . . . 21
3.5 $\boxed{con\,(cid) = con}$ extract concept name from a concept identifier . . . 21
3.6 $\boxed{con\,(cp) = con}$ extract concept name from a concept definition . . . 21
3.7 $\boxed{\overline{f^a}\,(cp) = \overline{f^a}}$ extract associated functions from a concept definition . . . 21
3.8 $\boxed{\overline{f^a}\,(C, cid) = \overline{f^a}}$ extract associated functions given a concept identifier . . . 21
3.9 $\boxed{\overline{fdef}\,(cm) = \overline{fdef}}$ extract function definitions from a concept map . . . 22
3.10 $\boxed{\overline{ty^a}\,(cp) = \overline{ty^a}}$ extract associated types from a concept definition . . . 22
3.11 $\boxed{\overline{ty^a}\,(C, cid) = \overline{ty^a}}$ extract associated types given a concept identifier . . . 22
3.12 $\boxed{\overline{tydef}\,(cm) = \overline{tydef}}$ extract type definitions from a concept map . . . 22
3.13 $\boxed{refined\,(C, cid) = \overline{cid}}$ concept identifiers refinement . . . 22
3.14 $\boxed{refined\,(C, con) = \overline{cid}}$ refined concepts of a concept . . . 23
3.15 $\boxed{overload\text{-}res\,(f\,\overline{\tau}, \overline{sig}) = sig?_{amb}}$ overload resolution . . . 23

4 Type System **24**
4.1 $\boxed{C \vdash cid \prec_1 \overline{cid}}$ direct concept refinement . . . 24
4.2 $\boxed{C \vdash cid \prec \overline{cid}}$ concept refinement . . . 24
4.3 $\boxed{C \vdash con \prec \overline{cid}}$ concept refinement (for concept names) . . . 24
4.4 $\boxed{find\text{-}scope\,(C, cid, \tau) = \tau?}$ type lookup, non-recursive . . . 24
4.5 $\boxed{find\text{-}rec\,(C, cid, \tau) = \tau?}$ recursive type lookup . . . 25
4.6 $\boxed{find\text{-}scope\,(C, cid, f) = \overline{sig}}$ non-recursive associated function lookup . . . 26
4.7 $\boxed{overload\text{-}set\,(f, \overline{f^a}) = \overline{sig}}$ generate overload set . . . 27
4.8 $\boxed{normalize\,(C, cid, \overline{sig}) = \overline{sig}'}$ fully qualify type names in an overload set . . . 27
4.9 $\boxed{find\text{-}rec\,(C, cid, f) = \overline{sig}}$ compute overload set recursively . . . 27
4.10 $\boxed{C\,;\,\Gamma\,;\,\overline{cid} \vdash e : \tau}$ constrained context typing . . . 28
4.11 $\boxed{tydef\,\surd}$ type definition well-formedness . . . 29
4.12 $\boxed{\overline{f^a}\,(C, cid) = \overline{f^a}}$ required associated functions . . . 29
4.13 $\boxed{\overline{ty^a}\,(C, cid) = \overline{ty^a}}$ required associated types . . . 29
4.14 $\boxed{\overline{sig}\,(C, M, cid?, cid) = \overline{sig}}$ defined function signatures (for concept maps) . . . 29

4.15 $\overline{fdef}\,(C,\ M,\ cid?,\ cid)\ =\ \overline{fdef}$ defined functions (for concept maps) 30
4.16 $\overline{tydef}\,(M,\ \overline{cid})\ =\ \overline{tydef}$ defined types (for concept maps) 30
4.17 $\overline{ty^a}\ \vdash\ \overline{tydef}\ \eqcirc\ \overline{tydef}^{\prec_1}$ one-level type definitions compatibility 31
4.18 $tydefs\text{-}check\,(C,\ cid,\ \overline{tydef})$ type definitions check 31
4.19 $tydefs_{\eqcirc}\,(C,\ M,\ cid,\ \overline{tydef})$ type definitions compatibility 31
4.20 $fdefs\text{-}check\,(C,\ cid,\ \overline{fdef})$ function definitions check 32
4.21 $fdefs_{\eqcirc}\,(C,\ M,\ cid_{cmap},\ cid?,\ \overline{fdef})$ function definitions compatibility 32
4.22 $icms\,(C,\ M,\ cid,\ cid?,\ \overline{tydef},\ \overline{fdef})\ =\ \overline{icm}'$ generate implicit concept maps 32
4.23 $C\,;\,M\,;\,T\ \vdash\ P\ \Downarrow\ C'\,;\,M'\,;\,T'$ program instantiation 33

5 Separate Type Checking Safety **34**
5.1 $P\ \Downarrow\ C\,;\,M\,;\,T$ user program instantiation 35

6 Statistics **35**

1 Introduction

The C++ language (ISO, 2003) is currently being extended by concepts (Gregor et al., 2006; Dos Reis and Stroustrup, 2006); we refer to the extended language as ConceptC++. Concepts provide an interface between *templates* and template arguments. Concept-constrained templates can be checked before they are used, in contrast to unconstrained templates in the current C++ that can only be fully checked when used.

This report describes a simplified version of the concept sublanguage that represents integral elements of ConceptC++. The purpose of the language is to capture the basic semantics of separate type checking with concepts as specified in (Gregor et al., 2008). This report provides the syntax and the semantics along with a low-level discussion; the overview, motivation, and high-level discussion are meant to be provided elsewhere. The abstract syntax definition and the semantic rules in this report are generated from a meta-description in Ott (Sewell et al., 2007), a tool for developing semantic definitions of programming languages. Semantic rules are given as, sometimes mutually-recursive, relations on the abstract syntax.

The features of C++ with concepts included in our formalization are:

- Concepts
  - refinement clauses
  - associated types
  - associated functions
- Templates
  - requirement clauses
  - single-expression bodies
- Concept maps
  - implementations for associated types and functions
- Auxiliary features
  - overload resolution
  - simple expressions

The following features are not supported:

- Template parameters
- Concept parameters
- Template concept maps
- Concept-based overloading
- Candidate set implementations of associated functions

2 Syntax

We describe the syntax of the core ConceptC++ language in the BNF form.

Some productions have annotations to the right of the right-hand side. The following annotations are understood by Ott.

- M indicates a metaproduction. These are not part of the free grammar for the relevant nonterminal, but instead are given meaning (in the theorem prover models) by translation into non-metaproductions. The translations, specified in the Ott source, are specific to each theorem prover.
- S is identical to M except that productions marked with S are admitted in parsing concrete terms.

The following annotations are for informational purposes only.

- [I] indicates a production that is not intended to be available in user programs but is useful in the metatheory.

- [L] indicates a library-implemented facility.

Since Ott currently does not provide a way to define functions (only relations are supported), some simple functions are embedded into the abstract syntax. Functions are usually indicated with function call syntax and a return type of the function is the non-terminal for which the function's production is given.

Note that some rules repeat production in other rules; this is intended and the repetition is captured by *subrule* constraints in section 2.2.

Also note that some terminals (see the "terminals" production in the grammar) are pretty printed identically to a non-terminal; however, they are not the same. It should be clear at the point of use if a particular symbol refers to a terminal or a non-terminal.

Furthermore, Ott synthesizes a `user_syntax` grammar (the last grammar rule) that lists all syntax available to the user.

2.1 Grammar Rules

typ_identifier	*id* in cpp standard, here type name
f	*id* in cpp standard, here function name
c	*id* in cpp standard, here concept name
tn	template name
var	variable names
v_{bool}	boolean value
v_{int}	integral value
i, j, k, n, m	index variables used in the grammar and the semantic rules

id_τ	::=			type identifier
	| *typ_identifier*			type name
	| $id_\tau\,(tydef)$	M	[I]	extract from $tydef$
$\overline{id_\tau}$	::=			type identifier sequence
	| $\overline{id_{\tau\,i}}^{\,i}$	M	[I]	many
	| $\overline{id_\tau}\,(\overline{ty^a})$	M	[I]	extract from $\overline{ty^a}$
	| $\overline{id_\tau}\,(\overline{tydef})$	M	[I]	extract from $\overline{tydef}$
con	::=			concept name
	| c			concept name
cid, cid_{cmap}, cid_{impl}	::=			concept identifier
	| con			concept name
	| $cid\,(cm)$	M	[I]	extract from cm
$cid?$	::=			concept identifier option
	| $None$	M	[I]	none
	| cid	M	[I]	some
	| $icm\,.\,cid$	M	[I]	extract from icm
$\overline{cid}$, $\overline{cid}^{\prec_1}$	::=			concept identifier sequence
	| cid	M	[I]	one
	| $\overline{\overline{cid_i}}^{\,i}$	M	[I]	many
	| $\{\, cid \in \overline{cid} \mid formula \,\}$	M	[I]	set comprehension for $\overline{cid}$
$scope$	::=			scope (used in qualified names)
	| con			concept name

	$\mid$ $cid\,(cm)$	M	[I]	extract from cm
req	$::=$			requirements clause
	$\mid$ `requires` $\overline{cid_i}^{\,i}$			def.
τ	$::=$			type specifier
	$\mid$ `int`			built-in
	$\mid$ `void`			built-in
	$\mid$ `bool`			built-in
	$\mid$ id_τ			type name
	$\mid$ $scope :: id_\tau$			qualified type name
$bintype$	$::=$			built-in types
	$\mid$ `int`			
	$\mid$ `void`			
	$\mid$ `bool`			
$\tau?$	$::=$			type specifier option
	$\mid$ $None$	M	[I]	none
	$\mid$ τ	M	[I]	some
	$\mid$ $\Gamma\,(var)$	M	[I]	variable environment lookup
$\overline{\tau}$	$::=$			type specifier sequence
	$\mid$ τ	M	[I]	one
	$\mid$ $\overline{\overline{\tau}_i}^{\,i}$	M	[I]	many
val	$::=$			values
	$\mid$ v_{int}			integer value
	$\mid$ v_{bool}			boolean value
	$\mid$ `obj` τ			object value
e	$::=$			expressions
	$\mid$ var			variable
	$\mid$ `obj` τ			object
	$\mid$ $f\,(\overline{e_i}^{\,i})$			application
	$\mid$ v_{int}			integral
	$\mid$ v_{bool}			boolean
	$\mid$ (e)	S		parenthesis
Γ	$::=$			type environment ($var \rightharpoonup \tau$)
	$\mid$ $\emptyset$	M	[I]	empty environment
	$\mid$ $[\overline{var_i : \tau_i}^{\,i}]$	M	[I]	convert function parameters to type environment
$tparam$	$::=$			type parameter

	$\mid$	$\texttt{typename}\, id_\tau$		def.
tl	$::=$			template
	$\mid$	$\texttt{template}\, req\, tn\, \{\, e \,\}$		def.
T	$::=$			templates environment
	$\mid$	$\emptyset$	M [I]	empty
	$\mid$	tl	M [I]	one
	$\mid$	$\overline{T_i}^{\,i}$	M [I]	flatten
ty^a	$::=$			associated type
	$\mid$	$tparam$		def.
$\overline{ty^a}, \overline{ty^a}^{\prec}$	$::=$			associated type sequence
	$\mid$	ty^a	M [I]	one
	$\mid$	$\overline{\overline{ty^a}_i}^{\,i}$	M [I]	many
f^a	$::=$			associated function
	$\mid$	$\tau\, f\, (\overline{\tau})$		return type, function name, parameters
$\overline{f^a}, \overline{f^a}^{\prec}$	$::=$			associated function sequence
	$\mid$	f^a	M [I]	one
	$\mid$	$\overline{\overline{f^a}_i}^{\,i}$	M [I]	many
cp	$::=$			concept
	$\mid$	$con\, \texttt{refines}\, \overline{cid}\, \{\, \overline{ty^a}\, \overline{f^a} \,\}$		def.
$cp?$	$::=$			concept option
	$\mid$	$None$	M [I]	none
	$\mid$	cp	M [I]	some
	$\mid$	$find\text{-}concept\, (C,\ con)$	M [I]	find a concept
C	$::=$			concept sequence
	$\mid$	$\emptyset$	M [I]	empty
	$\mid$	cp	M [I]	one
	$\mid$	$\overline{C_i}^{\,i}$	M [I]	many
sig	$::=$			function signature
	$\mid$	$\tau\, f\, (\overline{\tau})$		return type, name, parameters
	$\mid$	$sig\, (fdef)$	M [I]	from $fdef$
$sig?$	$::=$			function signature option
	$\mid$	$None$	M [I]	none

	$\mid$ sig	M	[I]	some
$sig?_{amb}$	$::=$			signature option or ambiguity
	$\mid$ ¿?		[I]	ambiguous
	$\mid$ $sig?$		[I]	sig option
$\overline{sig}, \overline{sig}^{\prec}$	$::=$			function signature sequence
	$\mid$ sig	M	[I]	one
	$\mid$ $\overline{\overline{sig}_i}^{\,i}$	M	[I]	flatten
	$\mid$ $rdup\,(\overline{sig})$	M	[I]	remove duplicates
	$\mid$ $\overline{sig}\,(\overline{fdef})$	M	[I]	from $\overline{fdef}$
	$\mid$ $\overline{sig}_1 \setminus \overline{sig}_2$	M	[I]	set difference
$fdef$	$::=$			function definition
	$\mid$ $\tau\, f\,(\overline{var_i : \tau_i}^{\,i})\,\{\, e \,\}$			return type, name, parameters, body
$fdef?$	$::=$			function declaration option
	$\mid$ $None$	M	[I]	none
	$\mid$ $fdef$	M	[I]	some
	$\mid$ $\overline{fdef} \vdash find\,(sig)$	M	[I]	find definition
$\overline{fdef}, \overline{fdef}^{\prec}$	$::=$			function definition sequence
	$\mid$ $fdef$	M	[I]	one
	$\mid$ $\overline{\overline{fdef}_i}^{\,i}$	M	[I]	many
	$\mid$ $\{\, fdef \in \overline{fdef} \mid formula \,\}$	M	[I]	set comprehension for $\overline{fdef}$
$tydef$	$::=$			type definition
	$\mid$ `typedef` $\tau\, id_\tau$			alias id_τ to τ
$\overline{tydef}, \overline{tydef}^{\prec_1}$	$::=$			type definition sequence
	$\mid$ $tydef$	M	[I]	one
	$\mid$ $\overline{\overline{tydef}_i}^{\,i}$	M	[I]	flatten
	$\mid$ $\{\, tydef \in \overline{tydef} \mid formula \,\}$	M	[I]	set comprehension
	$\mid$ $find\text{-}defs\,(id_\tau, \overline{tydef})$	M	[I]	find definitions for id_τ
$tydef?$	$::=$			type definition option
	$\mid$ $None$	M	[I]	none
	$\mid$ $tydef$	M	[I]	some
cm	$::=$			concept map
	$\mid$ `concept_map` $cid\,\{\, \overline{tydef}\,\overline{fdef} \,\}$			def.
	$\mid$ $icm\,.\,cm$	M	[I]	implicit concept map

$\overline{cm}$	::=			concept map sequence
	| cm	M	[I]	one
	| $\overline{\overline{cm}_i}^{\,i}$	M	[I]	many
	| $rm?\,(\overline{cm?})$	M	[I]	remove "none" options
$cm?$	::=			concept map option
	| $None$	M	[I]	none
	| cm	M	[I]	some
	| $cm\,(M,\ cid)$	M	[I]	find cid in M
$\overline{cm?}$	::=			concept map option sequence
	| $cm?$	M	[I]	one
	| $\overline{\overline{cm?}_i}^{\,i}$	M	[I]	many
icm	::=			implicit concept map
	| $(cm,\ cid?)$	M	[I]	concept map and optional "parent"
$\overline{icm}$	::=			implicit concept map sequence
	| icm	M	[I]	one
	| $\overline{\overline{icm}_i}^{\,i}$	M	[I]	flatten
	| $\{\, icm \in \overline{icm} \mid formula \,\}$	M	[I]	set comprehension
$icm?$	::=			implicit concept map option
	| $None$	M	[I]	none
	| icm	M	[I]	some
	| $cm\,(M,\ cid)$	M	[I]	find cid in M
M	::=			concept map environment
	| $\emptyset$	M	[I]	empty
	| icm	M	[I]	one
	| $\overline{M_i}^{\,i}$	M	[I]	flatten
d_{user}	::=			user definitions
	| cp			concept
	| tl			template
	| cm			concept map
P_{user}	::=			user syntax program
	| d_{user}	M		one
	| $\overline{P_{user}}_i^{\,i}$	M		flatten
d	::=			definitions
	| cp			concept
	| tl			template

	$\mid$	cm		concepts map
	$\mid$	icm	[I]	implicit map
P	$::=$			program
	$\mid$	d	M	one
	$\mid$	$\overline{P_i}^{\,i}$	M	flatten
$terminals$	$::=$			pretty printing of terminals
	$\mid$	$\triangleright$		gives
	$\mid$	(		
	$\mid$	)		
	$\mid$	$\mapsto$		maps to
	$\mid$	$\vdash$		context
	$\mid$	$\cup$		union
	$\mid$	::		scope operator
	$\mid$	`refines`		
	$\mid$	`requires`		
	$\mid$	`int`		
	$\mid$	`bool`		
	$\mid$	`void`		
	$\mid$	`template`		
	$\mid$	$\prec_1$		direct refinement
	$\mid$	$\prec$		refinement
	$\mid$	{		
	$\mid$	}		
	$\mid$	`concept_map`		
	$\mid$	`typedef`		
	$\mid$	`typename`		
	$\mid$	$\forall$		for all
	$\mid$	$\exists$		there exists
	$\mid$	$\in$		in
	$\mid$	$\notin$		not in
	$\mid$	$\vee$		or
	$\mid$	$\wedge$		and
	$\mid$	$\neq$		is not equal to
	$\mid$	,		separator (with correct spacing)
	$\mid$	[		left environment bracket
	$\mid$	]		right environment bracket
	$\mid$	:		types to
	$\mid$	¿?		ambiguity
	$\mid$	$\subseteq$		subset
	$\mid$	$\surd$		well-formed
	$\mid$	$\Downarrow$		instantiates to
	$\mid$	not defined in		
	$\mid$	defined in		
	$\mid$	$\setminus$		set minus for lists (preserves duplicates and order)

$\neg$	
$\mid$	
$\approx$	set equality on lists
$\emptyset$	empty
find-rec	
distinct	
type-decls-check	
fdefs-check	
$tydefs_{\doteq}$	
$fdefs_{\doteq}$	
$\doteq$	type definitions compatibility (one-level)
icms	
overload-res	
returns	
tydefs-check	
find-scope	
rdup	
con	
find-concept	
$\overline{f^a}$	
overload-set	
normalize	
refined	
find-rec	
rm?	remove *None* options
`obj`	object creation
None	empty option
$\overline{fdef}$	
$\overline{fdef}$	
cm	
$\overline{ty^a}$	extract required associated types for a concept and its refinements
$\overline{ty^a}$	extract associated types from a concept
$\overline{f^a}$	
id_τ	
$\overline{id_\tau}$	
$\overline{sig}$	extract function definitions from a concept map
$\overline{sig}$	extract function definitions from a concept map and its refinements
$\overline{tydef}$	
find-defs	
sig	
find-scope	
$\bullet$	separate a quantifier from a formula
fname	
params	
sig	
con	
$\overline{f^a}$	

	$\mid$	$\overline{ty^a}$		
	$\mid$	$\overline{tydef}$		
	$\mid$	$refined$		
	$\mid$	$find$		
	$\mid$	$length$		
	$\mid$	$matching$		
$formula$	$::=$			formulas that can be used in premises
	$\mid$	$judgement$		judgement
	$\mid$	$formula_1 \;\; .... \;\; formula_k$	M	[I] conjunction of formulas
	$\mid$	$(formula)$	M	[I] parenthesized
	$\mid$	$\neg\, formula$	M	[I] formula negation
	$\mid$	$cp \in P$	M	[I] concept definition in P
	$\mid$	$sig \in \overline{sig}$	M	[I] sig in $\overline{sig}$
	$\mid$	$tydef \in \overline{tydef}$	M	[I] type declaration in type declarations
	$\mid$	$\overline{tydef}_1 \;=\; \overline{tydef}_2$	M	[I] type declarations aliasing
	$\mid$	$\overline{tydef}_1 \;\approx\; \overline{tydef}_2$	M	[I] type declarations set equality
	$\mid$	$\overline{fdef}_1 \;=\; \overline{fdef}_2$	M	[I] function definitions aliasing
	$\mid$	$cp?_1 \;=\; cp?_2$	M	[I] concept option alias
	$\mid$	$tydef?_1 \;=\; tydef?_2$	M	[I] type declaration option equality
	$\mid$	$cm?_1 \;=\; cm?_2$	M	[I] concept map equality
	$\mid$	$cid?_1 \;=\; cid?_2$	M	[I] cid options equality
	$\mid$	$cid?_1 \;\neq\; cid?_2$	M	[I] concept id options inequality
	$\mid$	$\overline{cm} \;=\; \overline{cm}'$	M	[I] pointwise equality of concept map lists
	$\mid$	$cid \in \overline{cid}$	M	[I] cid is in $\overline{cid}$
	$\mid$	$id_\tau \in \overline{id_\tau}$	M	[I] list of $\overline{id_\tau}$ contains a type named id_τ
	$\mid$	$id_\tau \notin \overline{id_\tau}$	M	[I] id_τ not in $\overline{id_\tau}$
	$\mid$	$cid \;\neq\; cid'$	M	[I] concept id inequality
	$\mid$	$\overline{cid} \;=\; \overline{cid}'$	M	[I] concept id sets equality
	$\mid$	$\overline{f^a} \;=\; \overline{f^a}'$	M	[I] associated function sets equality
	$\mid$	$\overline{sig} \;=\; \overline{sig}'$	M	[I] function declarations equality
	$\mid$	$\overline{sig} \;\approx\; \overline{sig}'$	M	[I] function declarations set equality
	$\mid$	$\overline{\tau}_1 \;\approx\; \overline{\tau}_2$	M	[I] type specifiers set equality
	$\mid$	$\tau?_1 \;\neq\; \tau?_2$	M	[I] type specifier option inequality
	$\mid$	$\tau?_1 \;=\; \tau?_2$	M	[I] type specifier option alias
	$\mid$	$formula \;\wedge\; formula'$	M	[I] and
	$\mid$	$formula \;\vee\; formula'$	M	[I] or
	$\mid$	$\exists\, cid \in \overline{cid} \bullet formula$	M	[I] there exists concept id in a set of ids
	$\mid$	$\forall\, cid \in \overline{cid} \bullet formula$	M	[I] for all cid in cid set
	$\mid$	$\forall\, tydef \in \overline{tydef} \bullet formula$	M	[I] for all $tydef$ in $\overline{tydef}$ set
	$\mid$	$\forall\, fdef \in \overline{fdef} \bullet formula$	M	[I] for all $fdef$ in $\overline{fdef}$ set
	$\mid$	$distinct\,(\overline{sig})$	M	[I] distinct function signatures
	$\mid$	$distinct\,(\overline{id_\tau})$	M	[I] distinct type identifiers
	$\mid$	$e \;=\; e'$	M	[I] expression alias
	$\mid$	$\overline{sig} \;\subseteq\; \overline{sig}'\; matching\, f\, \overline{\tau}$	M	[I] $\overline{sig}$ in $\overline{sig}'$ matching call profile
	$\mid$	$length\, \overline{sig} >= n$	M	[I] $\overline{sig}$ length greater than n
	$\mid$	$length\, \overline{sig} \;=\; n$	M	[I] $\overline{sig}$ length equal n

$\mid$	$\overline{id_{\tau}}_1 \approx \overline{id_{\tau}}_2$	M	[I]	type ids set equality
$\mid$	$id_{\tau} = id_{\tau}{}'$	M	[I]	type identifier equality
$\mid$	$tl_1 = tl_2$	M	[I]	template equality
$\mid$	tl not defined in T	M	[I]	template not defined in templates
$\mid$	cp not defined in C	M	[I]	concept not defined in concepts
$\mid$	cid defined in C	M	[I]	a concept corresponding to cid exists in C
$\mid$	cm not defined in M	M	[I]	cm is not defined in M (i.e., no map for cid of cm)
$\mid$	$\eqcirc\,(tydef, \overline{tydef})$	M	[I]	definition $tydef$ is compatible with definitions $\overline{tydef}$
$\mid$	$icm?_1 = icm?_2$	M	[I]	equality of implicit concept map options
$\mid$	$rm?\,(\overline{icm?_i}^{\,i}) = \overline{icm}$	M	[I]	remove empty options
$\mid$	$rm?\,(\overline{\tau?_i}^{\,i}) = \overline{\tau}$	M	[I]	remove empty options
$\mid$	$\overline{icm}_1 = \overline{icm}_2$	M	[I]	compare lists of implicit concept maps

fdecls_name_def ::=
| $\mid$ $fname\,(sig) = f$ — extract function name from a function signature

fdecl_return_def ::=
| $\mid$ $returns\,(sig) = \tau$ — extract return type from a function signature

fdecl_params_def ::=
| $\mid$ $params\,(sig) = \overline{\tau}$ — extract parameters from a function signature

construct_function_def ::=
| $\mid$ $sig\,(f, \tau, \overline{\tau}) = sig$ — construct function signature

cid_name_def ::=
| $\mid$ $con\,(cid) = con$ — extract concept name from a concept identifier

concept_name_def ::=
| $\mid$ $con\,(cp) = con$ — extract concept name from a concept definition

concept_functions_def ::=
| $\mid$ $\overline{f^a}\,(cp) = \overline{f^a}$ — extract associated functions from a concept definition

cid_functions_def ::=
| $\mid$ $\overline{f^a}\,(C, cid) = \overline{f^a}$ — extract associated functions given a concept identifier

cmap_functions_def ::=
| $\mid$ $\overline{fdef}\,(cm) = \overline{fdef}$ — extract function definitions from a concept map

concept_fields_def ::=
| $\mid$ $\overline{ty^a}\,(cp) = \overline{ty^a}$ — extract associated types from a concept definition

cid_types_def ::=		
	$\overline{ty^a}\,(C,\ cid)\ =\ \overline{ty^a}$	extract associated types given a concept identifier
cmap_types_def ::=		
	$\overline{tydef}\,(cm)\ =\ \overline{tydef}$	extract type definitions from a concept map
cid_refined_cids_def ::=		
	$refined\,(C,\ cid)\ =\ \overline{cid}$	concept identifiers refinement
con_refined_cids_def ::=		
	$refined\,(C,\ con)\ =\ \overline{cid}$	refined concepts of a concept
find_fcall_fdecls_def ::=		
	$overload\text{-}res\,(f\,\overline{\tau},\ \overline{sig})\ =\ sig?_{amb}$	overload resolution
refines_dir_many ::=		
	$C\ \vdash\ cid\ \prec_1\ \overline{cid}$	direct concept refinement
cid_refines_many ::=		
	$C\ \vdash\ cid\ \prec\ \overline{cid}$	concept refinement
con_refines_many ::=		
	$C\ \vdash\ con\ \prec\ \overline{cid}$	concept refinement (for concept names)
find_type_def ::=		
	$find\text{-}scope\,(C,\ cid,\ \tau)\ =\ \tau?$	type lookup, non-recursive
find_type_rec_def ::=		
	$find\text{-}rec\,(C,\ cid,\ \tau)\ =\ \tau?$	recursive type lookup
find_fun_def ::=		
	$find\text{-}scope\,(C,\ cid,\ f)\ =\ \overline{sig}$	non-recursive associated function lookup
get_overset_def ::=		
	$overload\text{-}set\,(f,\ \overline{f^a})\ =\ \overline{sig}$	generate overload set
normalize_fdecls_def ::=		
	$normalize\,(C,\ cid,\ \overline{sig})\ =\ \overline{sig}'$	fully qualify type names in an overload set
find_fun_rec_def ::=		
	$find\text{-}rec\,(C,\ cid,\ f)\ =\ \overline{sig}$	compute overload set recursively

T_def ::=		
$\mid$	$C\,;\,\Gamma\,;\,\overline{cid} \vdash e : \tau$	constrained context typing
$typedecl_wf_def$ ::=		
$\mid$	$tydef\,\surd$	type definition well-formedness
$required_assfs_def$ ::=		
$\mid$	$\overline{f^a}\,(C,\ cid) = \overline{f^a}$	required associated functions
$required_assts_def$ ::=		
$\mid$	$\overline{ty^a}\,(C,\ cid) = \overline{ty^a}$	required associated types
$defined_fdecls_ref_def$ ::=		
$\mid$	$\overline{sig}\,(C,\ M,\ cid?,\ cid) = \overline{sig}$	defined function signatures (for concept maps)
$defined_fdefs_ref_def$ ::=		
$\mid$	$\overline{fdef}\,(C,\ M,\ cid?,\ cid) = \overline{fdef}$	defined functions (for concept maps)
$defined_assts_def$ ::=		
$\mid$	$\overline{tydef}\,(M,\ \overline{cid}) = \overline{tydef}$	defined types (for concept maps)
$assts_compat_def$ ::=		
$\mid$	$\overline{ty^a} \vdash \overline{tydef} \asymp \overline{tydef}^{\prec_1}$	one-level type definitions compatibility
$typedecls_check_def$ ::=		
$\mid$	$tydefs\text{-}check\,(C,\ cid,\ \overline{tydef})$	type definitions check
$typedecls_comatibility_def$ ::=		
$\mid$	$tydefs_{\asymp}\,(C,\ M,\ cid,\ \overline{tydef})$	type definitions compatibility
$fdefs_check_def$ ::=		
$\mid$	$fdefs\text{-}check\,(C,\ cid,\ \overline{fdef})$	function definitions check
$fdefs_compat_def$ ::=		
$\mid$	$fdefs_{\asymp}\,(C,\ M,\ cid_{cmap},\ cid?,\ \overline{fdef})$	function definitions compatibility
$implicit_cmaps_def$ ::=		
$\mid$	$icms\,(C,\ M,\ cid,\ cid?,\ \overline{tydef},\ \overline{fdef}) = \overline{icm}'$	generate implicit concept maps
p_inst_def ::=		
$\mid$	$C\,;\,M\,;\,T \vdash P \Downarrow C'\,;\,M'\,;\,T'$	program instantiation

$p_user_inst_def$ ::=
| $P \Downarrow C\,;\,M\,;\,T$ user program instantiation

$judgement$ ::=
| $fdecls_name_def$
| $fdecl_return_def$
| $fdecl_params_def$
| $construct_function_def$
| cid_name_def
| $concept_name_def$
| $concept_functions_def$
| $cid_functions_def$
| $cmap_functions_def$
| $concept_fields_def$
| cid_types_def
| $cmap_types_def$
| $cid_refined_cids_def$
| $con_refined_cids_def$
| $find_fcall_fdecls_def$
| $refines_dir_many$
| $cid_refines_many$
| $con_refines_many$
| $find_type_def$
| $find_type_rec_def$
| $find_fun_def$
| $get_overset_def$
| $normalize_fdecls_def$
| $find_fun_rec_def$
| T_def
| $typedecl_wf_def$
| $required_assfs_def$
| $required_assts_def$
| $defined_fdecls_ref_def$
| $defined_fdefs_ref_def$
| $defined_assts_def$
| $assts_compat_def$
| $typedecls_check_def$
| $typedecls_comatibility_def$
| $fdefs_check_def$
| $fdefs_compat_def$
| $implicit_cmaps_def$
| p_inst_def
| $p_user_inst_def$

$user_syntax$ ::=
| $typ_identifier$

$|\quad f$
$|\quad c$
$|\quad tn$
$|\quad var$
$|\quad v_{bool}$
$|\quad v_{int}$
$|\quad i$
$|\quad id_\tau$
$|\quad \overline{id_\tau}$
$|\quad con$
$|\quad cid$
$|\quad cid?$
$|\quad \overline{cid}$
$|\quad scope$
$|\quad req$
$|\quad \tau$
$|\quad bintype$
$|\quad \tau?$
$|\quad \overline{\tau}$
$|\quad val$
$|\quad e$
$|\quad \Gamma$
$|\quad tparam$
$|\quad tl$
$|\quad T$
$|\quad ty^a$
$|\quad \overline{ty^a}$
$|\quad f^a$
$|\quad \overline{f^a}$
$|\quad cp$
$|\quad cp?$
$|\quad C$
$|\quad sig$
$|\quad sig?$
$|\quad sig?_{amb}$
$|\quad \overline{sig}$
$|\quad fdef$
$|\quad fdef?$
$|\quad \overline{fdef}$
$|\quad tydef$
$|\quad \overline{tydef}$
$|\quad tydef?$
$|\quad cm$
$|\quad \overline{cm}$
$|\quad cm?$
$|\quad \overline{cm?}$
$|\quad icm$

| $\overline{icm}$
| $icm?$
| M
| d_{user}
| P_{user}
| d
| P
| $terminals$
| $formula$

2.2 Subrules

The above grammar is constrained by some *subrule* constraints that are checked by Ott. A subrule constraint $r1 \prec r2$ means that every production in $r1$ must also exist in $r2$ but $r2$ may have additional productions not in $r1$. The grammar above is constrained by the following subrule constraints:

- $val \prec e$,
- $bintype \prec \tau$,
- $f^a \prec sig$,
- $P_{user} \prec P$,
- $d_{user} \prec d$,
- $\overline{f^a} \prec \overline{sig}$,
- $cid \prec scope$.

3 Helper Judgements

This section contains the "helper" parts of the semantics, i.e., the simple but necessary parts. Each rule is listed in its own subsection. The type of the rule is given in a $\boxed{\text{box}}$. The type indicates which parts of the grammar are the "inputs" to a given rule. For example, the first rule in this section takes a signature *sig* and a function name f (*fname* and = are terminals). The type is followed by a brief comment. Next, the rules defining a relation on abstract syntax are given along with a short commentary.

3.1 $\boxed{fname\,(sig) = f}$ extract function name from a function signature

FDECL_NAME

$$\frac{}{fname\,(\tau\, f\,(\overline{\tau})) = f}$$

Extract the name of a function from a function signature.

3.2 $\boxed{returns\,(sig) = \tau}$ extract return type from a function signature

FDECL_RETURN

$$\frac{}{returns\,(\tau\, f\,(\overline{\tau})) = \tau}$$

Extract return type from a function signature.

3.3 $\boxed{params\,(sig) \;=\; \overline{\tau}}$ extract parameters from a function signature

FDECL_PARAMS

$$\frac{}{params\,(\tau\, f\,(\overline{\tau})) \;=\; \overline{\tau}}$$

Extract parameters from a function signature

3.4 $\boxed{sig\,(f,\, \tau,\, \overline{\tau}) \;=\; sig}$ construct function signature

CONSTRUCT_FUNCTION

$$\frac{}{sig\,(f,\, \tau,\, \overline{\tau}) \;=\; \tau\, f\,(\overline{\tau})}$$

Construct a function signature $\tau\, f\,(\overline{\tau})$ from its parts.

3.5 $\boxed{con\,(cid) \;=\; con}$ extract concept name from a concept identifier

CID_NAME

$$\frac{}{con\,(con) \;=\; con}$$

Given a concept identifier *con*, return *con*. In this version of the semantics a concept identifier is just a concept name; in future versions this rule may become more complex.

3.6 $\boxed{con\,(cp) \;=\; con}$ extract concept name from a concept definition

CONCEPT_NAME

$$\frac{}{con\,(con\,\texttt{refines}\,\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}) \;=\; con}$$

Given a concept $con\,\texttt{refines}\,\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}$, extract concept name *con*.

3.7 $\boxed{\overline{f^a}\,(cp) \;=\; \overline{f^a}}$ extract associated functions from a concept definition

CONCEPT_FUNCTIONS

$$\frac{}{\overline{f^a}\,(con\,\texttt{refines}\,\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}) \;=\; \overline{f^a}}$$

Given a concept $con\,\texttt{refines}\,\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}$, extract associated function $\overline{f^a}$.

3.8 $\boxed{\overline{f^a}\,(C,\, cid) \;=\; \overline{f^a}}$ extract associated functions given a concept identifier

CID_FUNCTIONS

$$\frac{\begin{array}{l}1.\; con\,(cid) \;=\; con \\ 2.\; find\text{-}concept\,(C,\, con) \;=\; con\,\texttt{refines}\,\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}\end{array}}{\overline{f^a}\,(C,\, cid) \;=\; \overline{f^a}}$$

Given an environment C and a concept identifier *cid*, extract associated functions of the concept designated by *cid*. First extract the concept name *con* from *cid* and then find the corresponding concept in C and

return its associated functions $\overline{f^a}$. Note that there is no error handling; if *cid* does not name a defined concept then the second premise cannot be fulfilled and this judgment does not apply.

3.9 $\boxed{\overline{fdef}\,(cm) \;=\; \overline{fdef}}$ extract function definitions from a concept map

CMAP_FUNCTIONS

$$\frac{}{\overline{fdef}\,(\texttt{concept_map}\; cid\,\{\,\overline{tydef}\;\overline{fdef}\,\}) \;=\; \overline{fdef}}$$

Given a concept map `concept_map` *cid* $\{\,\overline{tydef}\;\overline{fdef}\,\}$, return its associated function definitions $\overline{fdef}$.

3.10 $\boxed{\overline{ty^a}\,(cp) \;=\; \overline{ty^a}}$ extract associated types from a concept definition

CONCEPT_TYPES

$$\frac{}{\overline{ty^a}\,(con\;\texttt{refines}\;\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}) \;=\; \overline{ty^a}}$$

Given a concept *con* `refines` $\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}$, return its associated types $\overline{ty^a}$.

3.11 $\boxed{\overline{ty^a}\,(C,\; cid) \;=\; \overline{ty^a}}$ extract associated types given a concept identifier

CID_TYPES

$$\frac{\begin{array}{l}1.\; con\,(cid) \;=\; con\\ 2.\; \textit{find-concept}\,(C,\; con) \;=\; con\;\texttt{refines}\;\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}\end{array}}{\overline{ty^a}\,(C,\; cid) \;=\; \overline{ty^a}}$$

Given an environment C and a concept identifier *cid*, extract associated types of the concept designated by *cid*. First extract the concept name *con* from *cid* and then find the corresponding concept in C and return its associated types $\overline{ty^a}$. Note that there is no error handling; if *cid* does not name a defined concept then the second premise cannot be fulfilled and this judgment does not apply.

3.12 $\boxed{\overline{tydef}\,(cm) \;=\; \overline{tydef}}$ extract type definitions from a concept map

CMAP_TYPES

$$\frac{}{\overline{tydef}\,(\texttt{concept_map}\; cid\,\{\,\overline{tydef}\;\overline{fdef}\,\}) \;=\; \overline{tydef}}$$

Given an environment C and a concept identifier *cid*, extract associated types of the concept designated by *cid*. First extract the concept name *con* from *cid* and then find the corresponding concept in C and return its associated types $\overline{ty^a}$. Note that there is no error handling; if *cid* does not name a defined concept then the second premise cannot be fulfilled and this judgment does not apply.

3.13 $\boxed{refined\,(C,\; cid) \;=\; \overline{cid}}$ concept identifiers refinement

CRC_REF_CIDS

$$\frac{\begin{array}{l}1.\; con\,(cid) \;=\; con\\ 2.\; \textit{find-concept}\,(C,\; con) \;=\; con\;\texttt{refines}\;\overline{cid}\,\{\,\overline{ty^a}\;\overline{f^a}\,\}\end{array}}{refined\,(C,\; cid) \;=\; \overline{cid}}$$

A concept identifier designates a concept *use*. This judgment extracts concept identifiers for concepts that

a concept designated by a concept identifier *cid* refines. The concept designated by *cid* is looked up in the concept environment C and its refinements are transitively extracted and converted to appropriate concept identifiers $\overline{cid}$. In the current version of the semantics, $\overline{cid}$ is a sequence of concept names. Note that if *cid* names an undefined concept there is no explicit error handling, the judgment simply does not apply then.

3.14 $\boxed{refined\,(C,\ con)\ =\ \overline{cid}}$ refined concepts of a concept

CONRC_REF_CIDS

$$\frac{1.\ find\text{-}concept\,(C,\ con)\ =\ con\ \texttt{refines}\ \overline{cid}\,\{\,\overline{ty^a}\ \overline{f^a}\,\}}{refined\,(C,\ con)\ =\ \overline{cid}}$$

Given a concept name *con*, find the corresponding concept $con\ \texttt{refines}\ \overline{cid}\,\{\,\overline{ty^a}\ \overline{f^a}\,\}$ in the concepts environment C and extract its refined concepts $\overline{cid}$. The sequence $\overline{cid}$ actually names concept identifiers, i.e., a refinement clause refers to concept *uses*. In the current version of the semantics, $\overline{cid}$ is a sequence of concept names. Again, error handling is not explicit and the judgment does not apply if there is no concept named *con*.

3.15 $\boxed{overload\text{-}res\,(f\,\overline{\tau},\ \overline{sig})\ =\ sig?_{amb}}$ overload resolution

OVERLOAD_AMB

$$\frac{\begin{array}{l}1.\ distinct\,(\overline{sig})\\ 2.\ \overline{sig}\ \subseteq\ \overline{sig}\ matching\ f\,\overline{\tau}\\ 3.\ length\ \overline{sig} >= 2\end{array}}{overload\text{-}res\,(f\,\overline{\tau},\ \overline{sig})\ =\ ¿?}$$

Given a list of function signatures $\overline{sig}$ and a function call $f\overline{\tau}$ (f is a function name and $\overline{\tau}$ is a list of argument types) decide which of the function signatures in $\overline{sig}$ to use.

This rule (OVERLOAD_AMB) handles the ambiguous cases. First, ensure that there are no duplicate function signatures. Second, find all signatures that match the function call. Third, assert that there were at least two signatures that match; therefore, overload resolution concludes with ambiguity.

Note that *matching* is currently not specified in our semantics; it is just a placeholder for the appropriate matching strategy.

OVERLOAD_FOUND

$$\frac{\begin{array}{l}1.\ distinct\,(\overline{sig})\\ 2.\ \overline{sig}\ \subseteq\ \overline{sig}\ matching\ f\,\overline{\tau}\\ 3.\ length\ \overline{sig}\ =\ 1\\ 4.\ sig \in \overline{sig}\end{array}}{overload\text{-}res\,(f\,\overline{\tau},\ \overline{sig})\ =\ sig}$$

This rule (OVERLOAD_FOUND) handles the case where only one of the signatures in $\overline{sig}$ matches the call $f\overline{\tau}$. In this case, the matching function signature ("some" option) is returned as the result of overload resolution.

OVERLOAD_NOT_FOUND

$$\frac{\begin{array}{l}1.\ distinct\,(\overline{sig})\\ 2.\ \overline{sig}\ \subseteq\ \overline{sig}\ matching\ f\,\overline{\tau}\\ 3.\ length\ \overline{sig}\ =\ 0\end{array}}{overload\text{-}res\,(f\,\overline{\tau},\ \overline{sig})\ =\ None}$$

This rule handles the case where no function signatures match the function call. The "*None*" option is returned as the result of overload resolution.

4 Type System

4.1 $\boxed{C \vdash cid \prec_1 \overline{cid}}$ direct concept refinement

REFM_DIR

$$\frac{1.\ refined\,(C,\ cid)\ =\ \overline{cid}}{C \vdash cid \prec_1 \overline{cid}}$$

A concept designated by a concept identifier *cid directly* refines concepts designated by concept identifiers $\overline{cid}$, in the context of concept environment C. To find $\overline{cid}$, the helper relation *refined* is applied.

4.2 $\boxed{C \vdash cid \prec \overline{cid}}$ concept refinement

CIDREF_REC

$$\frac{\begin{array}{l} 1.\ C \vdash cid \prec_1 \overline{cid_i}^{\,i} \\ 2.\ C \vdash cid_i \prec \overline{cid_i}^{\,i} \end{array}}{C \vdash cid \prec \overline{cid_i\,\overline{cid_i}}^{\,i}}$$

The refinement relation is a transitive closure of the direct refinement relation. In the first premise, the sequence of concept identifiers for concepts directly refined by *cid* is found. Then, in premise 2, the refinement relation is applied recursively to every directly refined concept. The recursion terminates when a base case of a *cid* without direct refinements is reached. The result is a sequence containing every directly refined concept identifier cid_i and the concept identifiers $\overline{cid}_i$ that it refines.

4.3 $\boxed{C \vdash con \prec \overline{cid}}$ concept refinement (for concept names)

CONREF_REC

$$\frac{\begin{array}{l} 1.\ C \vdash cid \prec \overline{cid} \\ 2.\ con\,(cid)\ =\ con \end{array}}{C \vdash con \prec \overline{cid}}$$

The previous refinement judgments define refinement between concept *uses*, i.e., concept identifiers. Refinement also occurs between a concept and concept identifiers, when a concept is defined. This judgment defines the refinement relation between a concept name *con*, which refers to a concept definition, and concept identifiers $\overline{cid}$, which refer to concept uses in a refinement clause. The judgment finds an appropriate concept identifier *cid* (premise 2) and forwards the job to refinement between concept identifiers (premise 1).

4.4 $\boxed{find\text{-}scope\,(C,\ cid,\ \tau)\ =\ \tau?}$ type lookup, non-recursive

FT_BINTYPE

$$\frac{}{find\text{-}scope\,(C,\ cid,\ bintype)\ =\ bintype}$$

Built-in types are always in scope.

FT_QUALIFIED

$$\frac{\begin{array}{l} 1.\ con\,(cid)\ =\ con \\ 2.\ find\text{-}concept\,(C,\ con)\ =\ cp \\ 3.\ \overline{ty^a}\,(cp)\ =\ \overline{ty^a}\quad 4.\ id_\tau \in \overline{id_\tau}\,(\overline{ty^a}) \end{array}}{find\text{-}scope\,(C,\ cid,\ cid::id_\tau)\ =\ cid::id_\tau}$$

This rule describes successful qualified name lookup. The type identifier id_τ qualified with *cid* is looked up

in the scope of cid and in the environment C. The associated types of cid are extracted and if one of the associated types is named id_τ then it is returned as the result of the lookup. Note that the concept in which the type is looked up is the same as the concept with which the looked up name is qualified (both are cid).

FT_UNQUALIFIED

$$\frac{\begin{array}{l} 1.\ con\,(cid) = con \\ 2.\ \mathit{find\text{-}concept}\,(C,\ con) = cp \\ 3.\ \overline{ty^a}\,(cp) = \overline{ty^a} \\ 4.\ id_\tau \in \overline{id_\tau}\,(\overline{ty^a}) \end{array}}{\mathit{find\text{-}scope}\,(C,\ cid,\ id_\tau) = cid :: id_\tau}$$

This rule describes successful unqualified name lookup. The associated type named id_τ is looked up in the concept identified by cid.

FT_NONE_SCOPE

$$\frac{1.\ cid \neq cid'}{\mathit{find\text{-}scope}\,(C,\ cid,\ cid' :: id_\tau) = \mathit{None}}$$

This rule describes unsuccessful qualified name lookup. The type named id_τ, qualified by the concept identifier cid', is looked up in the concept identified by cid. The concept identifier cid' used to qualify id_τ does not match the identifier cid of the concept in which the lookup is being performed (premise 1) and the lookup returns None to indicate failure.

FT_NONE_QUALIFIED

$$\frac{\begin{array}{l} 1.\ con\,(cid) = con \\ 2.\ \mathit{find\text{-}concept}\,(C,\ con) = cp \\ 3.\ \overline{ty^a}\,(cp) = \overline{ty^a} \\ 4.\ id_\tau \notin \overline{id_\tau}\,(\overline{ty^a}) \end{array}}{\mathit{find\text{-}scope}\,(C,\ cid,\ cid :: id_\tau) = \mathit{None}}$$

This rule describes unsuccessful qualified name lookup. Here the qualifying scope and the concept in which the lookup is performed are the same (cid) but there is no associated type named id_τ (premise 4).

FT_NONE

$$\frac{\begin{array}{l} 1.\ con\,(cid) = con \\ 2.\ \mathit{find\text{-}concept}\,(C,\ con) = cp \\ 3.\ \overline{ty^a}\,(cp) = \overline{ty^a} \\ 4.\ id_\tau \notin \overline{id_\tau}\,(\overline{ty^a}) \end{array}}{\mathit{find\text{-}scope}\,(C,\ cid,\ id_\tau) = \mathit{None}}$$

This rule describes unsuccessful unqualified name lookup. Similarly to the previous rule, there is no associated type named id_τ.

4.5 $\boxed{\mathit{find\text{-}rec}\,(C,\ cid,\ \tau) = \tau?}$ recursive type lookup

FTREC_DIRECT

$$\frac{1.\ \mathit{find\text{-}scope}\,(C,\ cid,\ id_\tau) = \tau}{\mathit{find\text{-}rec}\,(C,\ cid,\ id_\tau) = \tau}$$

If the associated type is found in the current scope it is returned without looking any further.

FTREC_REC_SOME

$$\frac{\begin{array}{l} 1.\ \mathit{find\text{-}scope}\,(C,\ cid,\ id_\tau)\ =\ \mathit{None} \\ 2.\ C\ \vdash\ cid\ \prec_1\ \overline{cid_i}^{\,i} \\ 3.\ \overline{\mathit{find\text{-}rec}\,(C,\ cid_i,\ id_\tau)\ =\ \tau?_i}^{\,i} \\ 4.\ \mathit{rm?}\,(\overline{\tau?_i}^{\,i})\ =\ \tau\,\overline{\tau} \end{array}}{\mathit{find\text{-}rec}\,(C,\ cid,\ id_\tau)\ =\ \tau}$$

If the type named id_τ is not found in the current scope (premise 1) it is searched for in the directly refined concepts, which are extracted in premise . The search is performed recursively in the refined concept instances (premise 3) and empty options from the results are removed (premise 4). Note that type specifiers may repeat in the order in which they were reached from the current scope. Any shadowed identifiers will not be found because the search is terminated as soon as we find something.

FTREC_REC_NONE

$$\frac{\begin{array}{l} 1.\ \mathit{find\text{-}scope}\,(C,\ cid,\ id_\tau)\ =\ \mathit{None} \\ 2.\ C\ \vdash\ cid\ \prec_1\ \overline{cid_i}^{\,i} \\ 3.\ \overline{\mathit{find\text{-}rec}\,(C,\ cid_i,\ id_\tau)\ =\ \tau?_i}^{\,i} \\ 4.\ \mathit{rm?}\,(\overline{\tau?_i}^{\,i})\ = \end{array}}{\mathit{find\text{-}rec}\,(C,\ cid,\ id_\tau)\ =\ \mathit{None}}$$

Same as above but treats the case where no associated types named id_τ were found.

FTREC_REC_SCOPE

$$\frac{\begin{array}{l} 1.\ \mathit{find\text{-}rec}\,(C,\ cid',\ id_\tau)\ =\ \tau? \\ 2.\ C\ \vdash\ cid\ \prec\ \overline{cid_i}^{\,i} \\ 3.\ cid'\in\overline{cid_i}^{\,i}\ \vee\ cid'\ =\ cid \end{array}}{\mathit{find\text{-}rec}\,(C,\ cid,\ cid'::id_\tau)\ =\ \tau?}$$

The search for a qualified identifier is simply performed in the qualifying scope. Note that the scope must be either the current scope or one of the refined scopes. That requirements does not seem to be present in the concept wording (Gregor et al., 2008) although it may be implicit in clause 3.4.3.3 p1 because if one refers to a concept map in a constrained context there must be a corresponding concept map archetype.

4.6 $\boxed{\mathit{find\text{-}scope}\,(C,\ cid,\ f)\ =\ \overline{sig}}$ non-recursive associated function lookup

FIND_FUN_SCOPE

$$\frac{\begin{array}{l} 1.\ \mathit{con}\,(cid)\ =\ con \\ 2.\ \mathit{find\text{-}concept}\,(C,\ con)\ =\ cp \\ 3.\ \overline{f^a}\,(cp)\ =\ \overline{f^a} \\ 4.\ \mathit{overload\text{-}set}\,(f,\ \overline{f^a})\ =\ \overline{sig} \\ 5.\ \mathit{normalize}\,(C,\ cid,\ \overline{sig})\ =\ \overline{sig}' \end{array}}{\mathit{find\text{-}scope}\,(C,\ cid,\ f)\ =\ \overline{sig}'}$$

Given the concept environment C, a concept identifier cid, and a function name f, look up the set of functions named f in the concept identified by cid. First, a concept identified by cid is looked up (premise 1 and 2). The case where cid does not identify a defined concept, i.e., *find-concept* returns *None*, is not handled. Next, the associated functions $\overline{f^a}$ of the concept looked up in premise 2 are extracted. Then, the function signatures for functions named f are extracted into $\overline{sig}$. In premise 5, the signatures in $\overline{sig}$ are normalized, i.e., all type names are fully qualified. Finally, the normalized signatures $\overline{sig}'$ are returned.

4.7 $\boxed{\mathit{overload\text{-}set}\,(f,\ \overline{f^a})\ =\ \overline{\mathit{sig}}}$ generate overload set

G_OVER_EMPTY

$$\frac{}{\mathit{overload\text{-}set}\,(f,\ )\ =}$$

An empty set of associated functions produces an empty overload set.

G_OVER_REC

$$\frac{\begin{array}{l}1.\ \overline{f^a}\ =\ f^a\,\overline{f^a}' \\ 2.\ \mathit{fname}\,(f^a)\ =\ f \\ 3.\ \mathit{overload\text{-}set}\,(f,\ \overline{f^a}')\ =\ \overline{\mathit{sig}}\end{array}}{\mathit{overload\text{-}set}\,(f,\ \overline{f^a})\ =\ f^a\,\overline{\mathit{sig}}}$$

Examine each associated function in $\overline{f^a}$ by "calling" *overload-set* recursively. For every signature, check if the function is named f and if it is, add it to the result of the recursive overload set lookup.

4.8 $\boxed{\mathit{normalize}\,(C,\ \mathit{cid},\ \overline{\mathit{sig}})\ =\ \overline{\mathit{sig}}'}$ fully qualify type names in an overload set

N_FDECLS_EMPTY

$$\frac{}{\mathit{normalize}\,(C,\ \mathit{cid},\ )\ =}$$

Do not do anything for an empty overload set.

N_FDECLS_REC

$$\frac{\begin{array}{l}1.\ \overline{\mathit{sig}}\ \approx\ f^a\,\overline{\mathit{sig}}' \quad 2.\ \mathit{fname}\,(f^a)\ =\ f \\ 3.\ \mathit{returns}\,(f^a)\ =\ \tau \quad 4.\ \mathit{params}\,(f^a)\ =\ \overline{\tau} \\ 5.\ \mathit{find\text{-}rec}\,(C,\ \mathit{cid},\ \tau)\ =\ \tau' \\ 6.\ \mathit{find\text{-}rec}\,(C,\ \mathit{cid},\ \tau_1)\ =\ \tau_1'\ \ ..\ \ \mathit{find\text{-}rec}\,(C,\ \mathit{cid},\ \tau_n)\ =\ \tau_n' \\ 7.\ \mathit{normalize}\,(C,\ \mathit{cid},\ \overline{\mathit{sig}}')\ =\ \overline{\mathit{sig}}'' \\ 8.\ \mathit{sig}\,(f,\ \tau',\ \tau_1'\,..\,\tau_n')\ =\ f^{a\prime}\end{array}}{\mathit{normalize}\,(C,\ \mathit{cid},\ \overline{\mathit{sig}})\ =\ f^{a\prime}\,\overline{\mathit{sig}}''}$$

Given a concept environment C, a concept identifier *cid*, and an overload set $\overline{\mathit{sig}}$, fully qualify type names in each signature in $\overline{\mathit{sig}}$. A single signature is first extracted, in premise 1, from the overload set. Then it is disassembled into parts in premises 2–4. In premises 5–6, each type occurring in the signature is looked up in the concept identified by *cid*. In premises 7–8, the normalization process is called recursively for the remaining signatures and the current signature is reassembled, with the fully qualified types looked up earlier. The final result is the reassembled signature prepended to the result of the recursive normalization process.

4.9 $\boxed{\mathit{find\text{-}rec}\,(C,\ \mathit{cid},\ f)\ =\ \overline{\mathit{sig}}}$ compute overload set recursively

FIND_FUN_REC

$$\frac{\begin{array}{l}1.\ C\ \vdash\ \mathit{cid}\ \prec\ \overline{\mathit{cid}_i}^{\,i} \\ 2.\ \mathit{find\text{-}scope}\,(C,\ \mathit{cid},\ f)\ =\ \overline{\mathit{sig}} \\ 3.\ \overline{\mathit{find\text{-}scope}\,(C,\ \mathit{cid}_i,\ f)\ =\ \overline{\mathit{sig}_i}}^{\,i} \\ 4.\ \overline{\mathit{sig}}'\ =\ \overline{\mathit{sig}}\,\overline{\overline{\mathit{sig}_i}}^{\,i} \\ 5.\ \mathit{rdup}\,(\overline{\mathit{sig}}')\ =\ \overline{\mathit{sig}}''\end{array}}{\mathit{find\text{-}rec}\,(C,\ \mathit{cid},\ f)\ =\ \overline{\mathit{sig}}''}$$

Given the concept environment C and the concept identifier *cid*, look up the overload set for a function named f in the concept identified by *cid* and the concepts it refines. The set of concept identifers refined by *cid* is found in premise 1 and, in premises 2–3, the overload set for function f is looked up for each of these concepts. In premise 4, all of these overload sets are concatenated and in premise 5, duplicates (if there are any) are removed. The sequence of concept identifiers found in premise 1 maintains the depth-first order of discovery in traversal of refinement clauses. Furthermore, the *rdup* function in premise 5 keeps the first occurrence of each duplicate. Consequently, when there are duplicate signatures, the ones found first by depth-first traversal of the refinement hierarchy are kept.

4.10 $\boxed{C\,;\,\Gamma\,;\,\overline{cid} \vdash e : \tau}$ constrained context typing

T_INT

$$\frac{}{C\,;\,\Gamma\,;\,\overline{cid} \vdash v_{int} : \texttt{int}}$$

T_BOOL

$$\frac{}{C\,;\,\Gamma\,;\,\overline{cid} \vdash v_{bool} : \texttt{bool}}$$

Values of built in types have the corresponding built-in type.

T_VAR

$$\frac{1.\ \Gamma\,(var) = \tau}{C\,;\,\Gamma\,;\,\overline{cid} \vdash var : \tau}$$

Types of variables are looked up in the variable typing environment Γ.

T_OBJ

$$\frac{\begin{array}{l} 1.\ \overline{\mathit{find\text{-}rec}\,(C,\ cid_i,\ \tau) = \tau?_i}^{\,i} \\ 2.\ \mathit{rm?}\,(\overline{\tau?_i}^{\,i}) = \overline{\tau} \\ 3.\ \overline{\tau} \approx \tau \end{array}}{C\,;\,\Gamma\,;\,\overline{cid_i}^{\,i} \vdash \texttt{obj}\,\tau' : \tau}$$

This judgment gives the type of an object created by an object creation expression $\texttt{obj}\,\tau'$. To type the object, the type τ' must be looked up in the constraints $\overline{cid_i}^{\,i}$. In premise 1, the type τ' is looked up in each of the requirements and, in premise 2, the *None* results of unsuccessful searches are removed. Finally, the third premise asserts that the sequence of results $\overline{\tau}$ contains only one type τ (but possibly more than once). This type is the type of the object created by the object creation expression.

T_APP

$$\frac{\begin{array}{l} 1.\ \overline{\mathit{find\text{-}rec}\,(C,\ cid_i,\ f) = \overline{sig}_i}^{\,i} \\ 2.\ \mathit{distinct}\,(\overline{\overline{sig}_i}^{\,i}) \\ 3.\ \overline{C\,;\,\Gamma\,;\,\overline{cid_i}^{\,i} \vdash e_k : \tau_k}^{\,k} \\ 4.\ \mathit{overload\text{-}res}\,(f\,\overline{\tau_k}^{\,k},\ \overline{\overline{sig}_i}^{\,i}) = sig \\ 5.\ \mathit{returns}\,(sig) = \tau \end{array}}{C\,;\,\Gamma\,;\,\overline{cid_i}^{\,i} \vdash f\,(\overline{e_k}^{\,k}) : \tau}$$

This judgment types a function call $f\,(\overline{e_k}^{\,k})$. First, in premise 1, an overload set with functions named f is looked up; the second premise ensures that there are no duplicates in the set. In the third premise every argument is typed and overload resolution is performed in premise 4, finding the appropriate function signature *sig*. The return type extracted from that signature is the type of the function call. Note that if overload resolution is ambiguous or fails a function application cannot be typed.

4.11 $\boxed{tydef\ \surd}$ type definition well-formedness

TD__WFBINTYPE

$$\frac{}{\texttt{typedef}\ bintype\ id_\tau\ \surd}$$

The core language does currently not allow user-defined types. Therefore, the only types that can be used in a type declaration are built-in types.

4.12 $\boxed{\overline{f^a}\,(C,\ cid)\ =\ \overline{f^a}}$ required associated functions

ARF__REF

$$\frac{\begin{array}{l}1.\ C \vdash cid \prec \overline{cid_i}^{\,i} \\ 2.\ \overline{\overline{f^a}\,(C,\ cid_i)\ =\ \overline{f^a}_i}^{\,i} \\ 3.\ \overline{f^a}\,(C,\ cid)\ =\ \overline{f^a} \\ 4.\ normalize\,(C,\ cid,\ \overline{f^a})\ =\ \overline{f^a}'\end{array}}{\overline{f^a}\,(C,\ cid)\ =\ \overline{f^a}'\,\overline{\overline{f^a}_i}^{\,i}}$$

This judgment shows how associated functions are extracted from a concept identified by *cid* and from all the concepts that *cid* refines. In the first three premises, associated functions are extracted from *cid* and from the refined concepts and in the last premise the functions are normalized. Notice that a list of function declarations is returned and not a set. There may be duplicates but they cause no problems since name lookup will remove them.

4.13 $\boxed{\overline{ty^a}\,(C,\ cid)\ =\ \overline{ty^a}}$ required associated types

ARA__REF

$$\frac{\begin{array}{l}1.\ C \vdash cid \prec \overline{cid_i}^{\,i} \\ 2.\ \overline{\overline{ty^a}\,(C,\ cid_i)\ =\ \overline{ty^a}_i}^{\,i} \\ 3.\ \overline{ty^a}\,(C,\ cid)\ =\ \overline{ty^a}\end{array}}{\overline{ty^a}\,(C,\ cid)\ =\ \overline{ty^a}\,\overline{\overline{ty^a}_i}^{\,i}}$$

This judgment shows how required associated types are extracted for a concept. Note that all types are returned without qualification. That is, if a name is shadowed it will occur more than once in the result but it will be impossible to detect any difference between the occurrences.

4.14 $\boxed{\overline{sig}\,(C,\ M,\ cid?,\ cid)\ =\ \overline{sig}}$ defined function signatures (for concept maps)

DFDECLS__NONE

$$\frac{\begin{array}{l}1.\ C \vdash cid \prec \overline{cid_i}^{\,i} \quad 2.\ cid?\ =\ None \\ 3.\ \overline{cm\,(M,\ cid_i)\ =\ cm?_i}^{\,i} \\ 4.\ rm?\,(\overline{cm?_i}^{\,i})\ =\ \overline{cm_j}^{\,j} \\ 5.\ \overline{\overline{fdef}\,(cm_j)\ =\ \overline{fdef}_j}^{\,j} \\ 6.\ \overline{normalize\,(C,\ cid\,(cm_j),\ \overline{sig}\,(\overline{fdef}_j))\ =\ \overline{sig}_j}^{\,j}\end{array}}{\overline{sig}\,(C,\ M,\ cid?,\ cid)\ =\ \overline{\overline{sig}_j}^{\,j}}$$

This judgment extracts function signatures from concept maps for concept instances refining *cid*. The first rule handles the concept maps that have not been implicitly generated (premise 2). In premises 3–5, the concept maps for concepts refined by *cid* are looked up, and their function definitions are extracted. In the last premise, a signature of each definition is extracted and then normalized.

DFDECLS_SOME

$$\frac{\begin{array}{l} 1.\ C \vdash cid \prec \overline{cid_i}^{\,i} \quad 2.\ cid? = cid_{impl} \\ 3.\ \overline{cm\,(M,\ cid_i) = icm?_i}^{\,i} \\ 4.\ rm?\,(\overline{icm?_i}^{\,i}) = \overline{icm} \\ 5.\ \overline{icm_k}^{\,k} = \{\, icm \in \overline{icm} \mid icm\,.\,cid \neq cid_{impl} \,\} \\ 6.\ \overline{\overline{fdef}\,(icm_k\,.\,cm) = \overline{fdef}_k}^{\,k} \\ 7.\ \overline{normalize\,(C,\ cid\,(cm_k),\ \overline{sig}\,(\overline{fdef}_k)) = \overline{sig}_k}^{\,k} \end{array}}{\overline{sig}\,(C,\ M,\ cid?,\ cid) = \overline{\overline{sig}_k}^{\,k}}$$

The second rule is similar to the first but it handles the concept maps that have been implicitly defined (premise 2). In these cases, the concept maps for refined concepts that have been implicitly defined by the same source are ignored. In the third premise, implicit concept maps are searched for (the result of each search is an *icm*?); in the first rule, the implicit "parent" part of implicit concept maps was ignored (the result of each search was a *cm*?). The fifth premise in the second rule filters out all concept maps for refined concepts that have been implicitly defined by the same parent ($icm\,.\,cid \neq cid_{impl}$). This is done to ignore functions in refined concept maps that have exactly the same definitions since the definitions came from the same source: the "parent" concept map that caused implicit definition.

4.15 $\boxed{\overline{fdef}\,(C,\ M,\ cid?,\ cid) = \overline{fdef}}$ defined functions (for concept maps)

DFDEFS_NONE

$$\frac{\begin{array}{l} 1.\ C \vdash cid \prec \overline{cid_i}^{\,i} \\ 2.\ cid? = None \\ 3.\ \overline{cm\,(M,\ cid_i) = cm?_i}^{\,i} \\ 4.\ rm?\,(\overline{cm?_i}^{\,i}) = \overline{cm_j}^{\,j} \\ 5.\ \overline{\overline{fdef}\,(cm_j) = \overline{fdef}_j}^{\,j} \end{array}}{\overline{fdef}\,(C,\ M,\ cid?,\ cid) = \overline{\overline{fdef}_j}^{\,j}}$$

DFDEFS_SOME

$$\frac{\begin{array}{l} 1.\ C \vdash cid \prec \overline{cid_i}^{\,i} \quad 2.\ cid? = cid_{impl} \\ 3.\ \overline{cm\,(M,\ cid_i) = icm?_i}^{\,i} \\ 4.\ rm?\,(\overline{icm?_i}^{\,i}) = \overline{icm} \\ 5.\ \overline{icm_k}^{\,k} = \{\, icm \in \overline{icm} \mid icm\,.\,cid \neq cid_{impl} \,\} \\ 6.\ \overline{\overline{fdef}\,(icm_k\,.\,cm) = \overline{fdef}_k}^{\,k} \end{array}}{\overline{fdef}\,(C,\ M,\ cid?,\ cid) = \overline{\overline{fdef}_k}^{\,k}}$$

This judgment is very similar to the previous one but function definitions instead of signatures are extracted.

4.16 $\boxed{\overline{tydef}\,(M,\ \overline{cid}) = \overline{tydef}}$ defined types (for concept maps)

DEFASSTS

$$\frac{\begin{array}{l} 1.\ \overline{cm\,(M,\ cid_i) = cm?_i}^{\,i} \\ 2.\ rm?\,(\overline{cm?_i}^{\,i}) = \overline{cm_k}^{\,k} \\ 3.\ \overline{\overline{tydef}\,(cm_k) = \overline{tydef}_k}^{\,k} \end{array}}{\overline{tydef}\,(M,\ \overline{cid_i}^{\,i}) = \overline{\overline{tydef}_k}^{\,k}}$$

This judgment extracts definitions of associated types in concept maps for concepts identified by $\overline{cid_i}^{\,i}$. In contrast to the previous two judgments for associated function definitions, the concepts $\overline{cid_i}^{\,i}$ for which

definitions should be extracted are provided directly instead of a concept that they refine.

4.17 $\boxed{\overline{ty^a} \vdash \overline{tydef} \eqcirc \overline{tydef}^{\prec_1}}$ one-level type definitions compatibility

ASSTS_COMPAT

$$\frac{\begin{array}{l} 1.\ \overline{tydef}' = \{\, tydef \in \overline{tydef} \mid \neg\, id_\tau\,(tydef) \in \overline{id_\tau}\,(\overline{ty^a})\,\} \\ 2.\ \forall\, tydef \in \overline{tydef}' \bullet \mathit{find\text{-}defs}\,(id_\tau\,(tydef),\ \overline{tydef}^{\prec_1}) \approx tydef \end{array}}{\overline{ty^a} \vdash \overline{tydef} \eqcirc \overline{tydef}^{\prec_1}}$$

Compatibility of type declarations is decided in light of the associated types of the concept for which the concept map is defined. The rule consists of two parts:

1. Narrow the set of type declarations to those that are not for one of the associated types.
2. Make sure that the type declarations provided for a new concept map are compatible with the existing declarations.

The compatibility of "siblings," i.e., type declarations in $\overline{tydef}^{\prec_1}$ does not have to be checked separately. For non-shadowed type names, line 2 ensures sibling compatibility through the new definition in $\overline{tydef}$. For shadowed names, compatibility was already ensured when these names were defined. In particular, definitions for shadowed names may differ if the sources of a name in the refinement hierarchy are different.

4.18 $\boxed{\mathit{tydefs\text{-}check}\,(C,\ cid,\ \overline{tydef})}$ type definitions check

TYPEDECLS_CHECK

$$\frac{\begin{array}{l} 1.\ \overline{ty^a}\,(C,\ cid) = \overline{ty^a}^{\prec} \\ 2.\ \overline{id_\tau}\,(\overline{tydef}) \approx \overline{id_\tau}\,(\overline{ty^a}^{\prec}) \\ 3.\ \forall\, tydef \in \overline{tydef} \bullet tydef\ \surd \\ 4.\ \mathit{distinct}\,(\overline{id_\tau}\,(\overline{tydef})) \end{array}}{\mathit{tydefs\text{-}check}\,(C,\ cid,\ \overline{tydef})}$$

This relation is a helper relation used in the P_INST_CMAP rule. When processing a concept map, it must be checked that there is a type declaration corresponding to every associated type in the current concept instance (represented by *cid*). Also, there cannot be more than one type declaration for each associated type (the distinctiveness check) and every type declaration must be well-formed.

4.19 $\boxed{\mathit{tydefs}_{\eqcirc}\,(C,\ M,\ cid,\ \overline{tydef})}$ type definitions compatibility

TYPEDECLS_COMPAT

$$\frac{\begin{array}{l} 1.\ \overline{ty^a}\,(C,\ cid) = \overline{ty^a} \\ 2.\ C \vdash cid \prec_1 \overline{cid}^{\prec_1} \\ 3.\ \overline{tydef}\,(M,\ \overline{cid}^{\prec_1}) = \overline{tydef}^{\prec_1} \\ 4.\ \overline{ty^a} \vdash \overline{tydef} \eqcirc \overline{tydef}^{\prec_1} \end{array}}{\mathit{tydefs}_{\eqcirc}\,(C,\ M,\ cid,\ \overline{tydef})}$$

This judgment is a helper judgment used in the P_INST_CMAP rule. First (premise 1), the associated types required by the concept instance represented by *cid* are extracted; no associated type requirements from refined concepts are included. Next (premise 2), concepts instances directly refined by *cid* are extracted into $\overline{cid}^{\prec_1}$. Then (premise 3), all type definitions provided by concept maps for concept identifiers in $\overline{cid}^{\prec_1}$ are extracted into $\overline{tydef}^{\prec_1}$. It is important to remember that each concept map must provide type definitions for all associated type requirements reached through refinement. Since type definitions in $\overline{tydef}^{\prec_1}$ were extracted from existing concept maps, these were already checked for compatibility conflicts and inconsistencies. For concept instances in $\overline{cid}^{\prec_1}$ for which concept maps do not exist yet, implicit concept maps will be created,

assuming that the currently processed definition passes compatibility and consistency checks. The implicit concept maps will propagate the definitions from the currently processed map and if an incompatibility is introduced, it will be detected later as implicit concept maps are created. Finally (premise 4), the one-level compatibility check is performed between type definitions in $\overline{tydef}$ and in $\overline{tydef}^{\prec_1}$.

4.20 $\boxed{fdefs\text{-}check\,(C,\ cid,\ \overline{fdef})}$ function definitions check

FDEFS_CHECK

$$\frac{\begin{array}{l}1.\ \forall\,\tau\,f\,(\overline{var_i : \tau_i}^{\,i})\,\{\,e\,\} \in \overline{fdef} \bullet C\,;\,[\overline{var_i : \tau_i}^{\,i}]\,;\,cid \vdash e : \tau \\ 2.\ distinct\,(\overline{sig}\,(\overline{fdef}))\end{array}}{fdefs\text{-}check\,(C,\ cid,\ \overline{fdef})}$$

This judgment is a helper judgment used in the P_INST_CMAP rule. First (premise 1), all function definitions are type checked. After the definitions are type checked, a distinctiveness check is performed to ensure that there are no conflicting definitions. Which definitions should be provided is checked in the FDEFS_COMPATIBILITY rule.

4.21 $\boxed{fdefs_{=}\,(C,\ M,\ cid_{cmap},\ cid?,\ \overline{fdef})}$ function definitions compatibility

FDEFS_COMPAT

$$\frac{\begin{array}{l}1.\ \overline{f^a}\,(C,\ cid) = \overline{f^a}^{\prec} \\ 2.\ \overline{sig}\,(C,\ M,\ cid?,\ cid) = \overline{sig}^{\prec} \\ 3.\ distinct\,(\overline{sig}^{\prec}) \\ 4.\ \overline{sig}\,(\overline{fdef}) \approx \overline{f^a}^{\prec} \setminus \overline{sig}^{\prec}\end{array}}{fdefs_{=}\,(C,\ M,\ cid,\ cid?,\ \overline{fdef})}$$

This judgment is a helper judgment used in the P_INST_CMAP rule. First (premise 1), all required associated functions are collected; both for the concept instance represented by *cid* and for any concept instance that *cid* refines. Next (premise 2), all function definitions for concept maps of the refined concepts are collected. Then (premise 3), a distinctiveness check is performed on the function definitions from the concept maps for the refined concepts. Finally (premise 4), there must be a function definition for every associated function except those that are currently defined. Since the $\overline{f^a}$ function normalizes associated functions (i.e., fully qualifies every name), function definitions must be provided using fully qualified associated type names. This is a restriction compared to the ConceptC++ language; it does not reduce expressive power of the language but prevents certain errors from ever occurring.

4.22 $\boxed{icms\,(C,\ M,\ cid,\ cid?,\ \overline{tydef},\ \overline{fdef}) = \overline{icm}'}$ generate implicit concept maps

IMPLICIT_CMAPS_SOME

$$\frac{\begin{array}{l}1.\ cid? = cid_{impl} \qquad 2.\ C \vdash cid \prec_1 \overline{cid}^{\prec_1} \\ 3.\ \overline{cid_i}^{\,i} = \{\,cid \in \overline{cid}^{\prec_1} \mid cm\,(M,\ cid) = None\,\} \\ 4.\ \overline{fdef}\,(C,\ M,\ cid_{impl},\ cid) = \overline{fdef}^{\prec} \\ 5.\ \overline{\overline{ty^a}\,(C,\ cid_i) = \overline{ty^a}^{\prec}{}_i}^{\,i} \quad 6.\ \overline{\overline{f^a}\,(C,\ cid_i) = \overline{f^a}^{\prec}{}_i}^{\,i} \\ 7.\ \overline{\overline{tydef}_i = \{\,tydef \in \overline{tydef} \mid id_\tau\,(tydef) \in \overline{id_\tau}\,(\overline{ty^a}^{\prec}{}_i)\,\}}^{\,i} \\ 8.\ \overline{\overline{fdef}_i = \{\,fdef \in \overline{fdef}\,\overline{fdef}^{\prec} \mid sig\,(fdef) \in \overline{f^a}^{\prec}{}_i\,\}}^{\,i} \\ 9.\ \overline{cm_i = \texttt{concept_map}\ cid_i\,\{\,\overline{tydef}_i\,\overline{fdef}_i\,\}}^{\,i}\end{array}}{icms\,(C,\ M,\ cid,\ cid?,\ \overline{tydef},\ \overline{fdef}) = \overline{(cm_i,\ cid_{impl})}^{\,i}}$$

This judgment shows how implicit concept maps are generated. The "inputs" are the concepts environment C, the concept maps environment M, the concept identifier *cid* of the concept map that initiated generation

process, the potential "parent" concept map $cid?$, and the type ($\overline{tydef}$) and function ($\overline{fdef}$) definitions from the concept map for the concept identified by cid. The "result" of the judgment is a set of generated concept maps. The first rule shows generation for concept maps that have a "parent" and the second rule shows generation for concept maps that do not have one (premise 1 in both rules). The second premise extracts identifiers of the concepts directly refined by the concept identified by cid and the third premise keeps only the ones for which no concept map exists yet. Premise 4 finds function definitions in concept maps for concepts refined by cid; these are used later in premise 8. In premises 5–6, required associated types and functions are extracted for each of the directly refined concepts. In premises 7–8, the definitions for associated functions and types are taken from the available definitions. For associated types, the concept map for cid is guaranteed to contain a definition for every associated type in each refined concept, as described earlier. For associated functions, the definitions are taken from the current concept map and from the refined concept maps ($\overline{fdef}\,\overline{fdef}^{\prec}$). According to the judgments described earlier, this sequence of functions is guaranteed to contain a definition for every associated function in refined concepts that is still undefined. Finally, premise 9 assembles the definitions into concept maps. Note that in the first rule the returned concept maps are constructed with cid_{impl} as the "parent" (in the conclusion of the rule), while in the second rule cid is set to be the "parent." This is because in the first rule the concept map for cid was itself implicitly defined and it simply propagates the "parent," while in the second rule the concept map for cid is explicitly defined and it is the "parent" itself.

IMPLICIT_CMAPS_NONE

$$\frac{\begin{array}{l}
1.\ cid? = None \quad 2.\ C \vdash cid \prec_1 \overline{cid}^{\prec_1} \\
3.\ \overline{cid_i}^{\,i} = \{\, cid \in \overline{cid}^{\prec_1} \mid cm\,(M,\, cid) = None \,\} \\
4.\ \overline{fdef}\,(C,\, M,\, None,\, cid) = \overline{fdef}^{\prec} \\
5.\ \overline{\overline{ty^a}\,(C,\, cid_i) = \overline{ty^a}^{\prec}{}_i}^{\,i} \quad 6.\ \overline{\overline{f^a}\,(C,\, cid_i) = \overline{f^a}^{\prec}{}_i}^{\,i} \\
7.\ \overline{\overline{tydef}_i = \{\, tydef \in \overline{tydef} \mid id_\tau\,(tydef) \in \overline{id_\tau}\,(\overline{ty^a}^{\prec}{}_i) \,\}}^{\,i} \\
8.\ \overline{\overline{fdef}_i = \{\, fdef \in \overline{fdef}\,\overline{fdef}^{\prec} \mid sig\,(fdef) \in \overline{f^a}^{\prec}{}_i \,\}}^{\,i} \\
9.\ \overline{cm_i = \texttt{concept_map}\ cid_i\,\{\, \overline{tydef}_i\,\overline{fdef}_i \,\}}^{\,i}
\end{array}}{icms\,(C,\, M,\, cid,\, cid?,\, \overline{tydef},\, \overline{fdef}) = \overline{(cm_i,\, cid)}^{\,i}}$$

4.23 $\boxed{C\,;\,M\,;\,T \vdash P \Downarrow C'\,;\,M'\,;\,T'}$ program instantiation

P_INST_NIL

$$\frac{}{C\,;\,M\,;\,T \vdash \Downarrow C\,;\,M\,;\,T}$$

This judgment processes a meta-program, definition by definition. The check is performed in the environment $C\,;\,M\,;\,T$ where C is the sequence of all defined concepts, M is the sequence of all defined concept maps, and T is the sequence of all defined templates. The program P produces a new environment $C'\,;\,M'\,;\,T'$ if it is correct—every definition in P extends the appropriate part of the environment.

The first rule handles empty programs: nothing has to be done.

P_INST_TMPL

$$\frac{\begin{array}{l}
1.\ tl = \texttt{template}\,\texttt{requires}\,\overline{cid_i}^{\,i}\ tn\,\{\, e \,\} \\
2.\ \overline{cid_i \text{ defined in } C}^{\,i} \\
3.\ tl \text{ not defined in } T \\
4.\ C\,;\,\emptyset\,;\,\overline{cid_i}^{\,i} \vdash e : \tau \\
5.\ C\,;\,M\,;\,tl\,T \vdash P \Downarrow C'\,;\,M'\,;\,T'
\end{array}}{C\,;\,M\,;\,T \vdash tl\,P \Downarrow C'\,;\,M'\,;\,T'}$$

The second rule handles template definitions. The first premise simply creates an alias for a template given as the "input" in the conclusion so that parts of the template can be accessed. The second premise checks that concepts used in the refinement clause of the template are defined in the environment and the third rule checks that the template named tn is not already defined. In premise 4, the expression e contained in the

template is type checked given C, an empty variable environment $\emptyset$, and the requirements of the template $\overline{cid_i}^{\,i}$. Finally, the last premise asserts that the rest of the program evaluates to a new environment given the original environment extended with the newly defined template.

$$\frac{\begin{array}{l}\text{P__INST__CONCEPT}\\ 1.\ cp \text{ not defined in } C\\ 2.\ cp\, C\,;\, M\,;\, T \vdash P \Downarrow C'\,;\, M'\,;\, T'\end{array}}{C\,;\, M\,;\, T \vdash cp\, P \Downarrow C'\,;\, M'\,;\, T'}$$

Currently, there are is no checks for concept definitions beyond ensuring that a concept is not being redefined (premise 1).

$$\frac{\begin{array}{l}\text{P__INST__CMAP}\\ 1.\ C\,;\, M\,;\, T \vdash (cm,\, None)\, P \Downarrow C'\,;\, M'\,;\, T\end{array}}{C\,;\, M\,;\, T \vdash cm\, P \Downarrow C'\,;\, M'\,;\, T}$$

Checking of a concept map is forwarded to the next rule for checking of implicit concept maps. The "input" concept map cm, given in the conclusion, is used to construct an implicit concept map $(cm,\, None)$, which is checked by the rule for implicit concept maps in the only premise.

$$\frac{\begin{array}{l}\text{P__INST__CMAP__IMPL}\\ 1.\ icm = (cm,\, cid?)\\ 2.\ cm = \texttt{concept_map}\, cid\, \{\, \overline{tydef}\, \overline{fdef}\, \}\\ 3.\ cm \text{ not defined in } M\\ 4.\ tydefs\text{-}check\,(C,\, cid,\, \overline{tydef})\\ 5.\ tydefs_{=}\,(C,\, M,\, cid,\, \overline{tydef})\\ 6.\ fdefs\text{-}check\,(C,\, cid,\, \overline{fdef})\\ 7.\ fdefs_{=}\,(C,\, M,\, cid,\, cid?,\, \overline{fdef})\\ 8.\ icms\,(C,\, M,\, cid,\, cid?,\, \overline{tydef},\, \overline{fdef}) = \overline{icm_i}^{\,i}\\ 9.\ C\,;\, icm\, M\,;\, T \vdash \overline{icm_i}^{\,i}\, P \Downarrow C'\,;\, M'\,;\, T'\end{array}}{C\,;\, M\,;\, T \vdash icm\, P \Downarrow C'\,;\, M'\,;\, T'}$$

The last rule processes implicit concept maps. The first premise simply creates an alias to the implicit concept map given as an "input" and the second premise creates an alias to the concept map part of the implicit concept map tuple. The third premise asserts that cm does not redefine an existing concept map. The following 5 premises perform, in order, checking of type definitions, compatibility check of type definitions, function definitions check, function definitions compatibility check, and generation of the implicit concept map. All of these tasks have been described earlier in this document. The last premise processes the rest of the program.

5 Separate Type Checking Safety

The goal of separate type checking is to guarantee that templates can be checked against their requirements at the time of definition. Then, definitions in concept maps are checked separately. Finally, when using a template, it only has to be checked that the necessary concept maps exist. In our core language, the only task remaining is to substitute definitions but no type checking is required: the previous checks guarantee that type checking succeeds. The details of separate type checking are given in (Zalewski and Schupp, 2008), here we give only a brief overview.

Assume $\emptyset\,;\, \emptyset\,;\, \emptyset \vdash P \Downarrow C'\,;\, M'\,;\, T'$

Then: For any template $tmpl \in T$, if concept maps enumerated in the requirement clause of the template exist, then the template expression is guaranteed to type check after type definitions from concept maps have been substituted for type names used in the expression (type checking succeeds in the context of function definitions given in concept maps).

This is an informal statement of the separate type checking safety property. The following properties are implied:

- No Argument-Dependent Lookup (ADL) is necessary. All names are resolved during type checking of templates and then bound to the implementations provided in concept maps.

- No errors can occur during template instantiation if the required concept maps are defined.
- Programs developed independently against a common set of concepts are guaranteed to type check when used together.

In our restricted language, separate type checking safety is indeed guaranteed for all cases yet in C++ there are some cases where separate type checking does not exclude errors at template instantiation time. Järvi et al. (2006) discuss in detail how separate type checking may fail, due to concept-based overloading. Our core language investigates other aspects of concepts design, in particular we concentrate on name lookup and refinement. Since we do not include the features that may break the safety of separate type checking, template instantiation never results in errors.

5.1 $\boxed{P \Downarrow C\,;\,M\,;\,T}$ user program instantiation

$$\frac{1.\ \emptyset\,;\,\emptyset\,;\,\emptyset \vdash P_{user} \Downarrow C\,;\,M\,;\,T}{P_{user} \Downarrow C\,;\,M\,;\,T}\ \text{P__USER__INST}$$

6 Statistics

```
Definition rules:        62 good    0 bad
Definition rule clauses: 229 good    0 bad
```

References

D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *Proc. 21st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 291–310, 2006.

D. Gregor, B. Stroustrup, J. Widman, and J. Siek. Proposed wording for concepts (revision 5). Technical Report N2617=08-0127, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2008.

ISO/IEC 14882 Int. Standard for Information Systems – Programming Languages – C++ . ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2nd edition, 2003.

J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. Siek. Algorithm specialization in generic programming: challenges of constrained generics in C++. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 272–282. ACM, 2006.

G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *Proc. 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 295–308. ACM, 2006.

P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: effective tool support for the working semanticist. In *Proc. ACM SIGPLAN Internat. Conf. on Functional Programming (ICFP)*, pages 1–12. ACM, 2007.

M. Zalewski and S. Schupp. A semantic definition of separate type checking in C++ with concepts. *Journal of Object Technology*, 2008. Submitted.