Typing and Subtyping for an Object-Oriented Process Algebra

Malcolm Tyrrell* and Andrew Butterfield
Dept. of Computer Science, Trinity College Dublin

February 18, 2002

Abstract

Oompa is an extension of the π-calculus which provides class-based object-orientation in the style of the CORBA object model. This technical report supplements the earlier presentation of Oompa [TBD00] by providing a sound type system with structural subtyping. The classes and interfaces of class-based languages typically give rise to highly inter-referential, and sometimes self-referential types. A key feature of our system is that these types are transformed into a bound recursive form independent of other definitions, and our subtyping algorithm guarantees termination upon types of this form.

1 Introduction

Object-based distributed systems, such as CORBA [OMG98] systems, are extremely complex and can contain instances of more than one software paradigm. Consequently, an approach to modelling such a system should not attempt to capture all of its behaviours — it should focus on certain aspects and abstract away from others. We consider the interesting behaviours of object-based distributed systems to be at and above the level of objects. Thus, the underlying mechanisms that provide the language-level object-orientation are less of a concern. The CORBA object model provides a standard view of such behaviour for the software entities participating in a CORBA system. The formalism we provide embodies this approach to object-orientation, allowing CORBA systems to be modelled simply and naturally.

*Supported by Enterprise Ireland Basic Research Grant No. SC/97/631

Our formalism is built upon the π-calculus [Mil99], augmenting it with class-based object orientation. This technical report introduces a type system for Oompa, and it is interesting to note that both the channel mechanisms and the object-orientation present challenges. In the case of channels, we need recursive types to provide sufficiently dynamic behaviour for the kinds of systems we are modelling. In the case of objects, the typically inter-referential nature of their types (as defined by interface and class definitions) can lead to an infinite sequence of definition look-ups. These become serious issues when it comes to subtyping, and a terminating algorithm requires a careful approach.

In the style of [AC91], we introduce a notion of type tree and using an algorithm based on [PS93] we can deal with types not involving definitions. We deal with the issue of definitions by defining a function, similar to one in [AC91], which can remove the dependency of types on definitions. Our subtype algorithm is shown to be terminating and both complete and sound relative to a tree semantics. As regards Oompa, the important result is that an Oompa expression which passes our type safety system never commits a type violation.

This technical report is structured into 8 sections. Following this introductory section, Section 2 presents the syntax and operational semantics of our calculus, Oompa. Section 3 describes the basic type system. Section 4 considers Oompa type trees which are introduced to deal with recursive types. In Section 5, we define a tree simulation between Oompa type trees and introduce our subtyping algorithm, relating the two with soundness and completeness results. In Section 6 we prove that our type safety system prevents type violations using a notion of dynamic type safety. We relate our work to others in Section 7 and discuss conclusions and future work in Section 8. Appendix A contains some definitions we choose to leave out of the main text.

2 The Oompa Calculus

The π-calculus provides a suitable starting point for modelling dynamic communicating systems. It supports concurrent processes, channel creation, channel passing communication and is computationally powerful. However, for modelling the behaviour we are interested in, it is too high-level in the sense that its processes are much more abstract entities than CORBA objects, and too low-level in the sense that to model such objects would require a lot of complexity.

Taking the π-calculus as a base, we introduce a calculus which has objects

whose method bodies contain π-calculus-like code. Like processes of the π-calculus, a running method can fork, create channels and engage in channel-based communication. Moreover, in our system they can perform operations at the object level, by creating new objects, invoking methods and using object state.

2.1 Relationship to the CORBA Object Model

Oompa is intended to model CORBA systems at the object level and above. Consequently, we provide a calculus which embodies an approach close to CORBA's view of objects and avoid the complexity of dealing with encodings of object behaviour later in our modelling work. The CORBA approach to objects is called the *CORBA Object Model* [OMG98], and represents an attempt to capture commonality across object systems and programming languages. Consequently, it is very abstract and quite general.

The Object Model describes a system which contains *data*, *code* and *execution engines*. The role of the execution engine is to interpret the code and perform the described computation, possibly altering some of the data. CORBA considers this process as the performance of a *service*, and perhaps the most significant feature of the Object Model is that it provides a standard way of requesting that these services be performed. Services are grouped into *objects* and clients are given an *interface* which is a description of how to make requests on the services offered by an object.

It is at this interface level that we choose to view CORBA systems, so we bring the interfaces and static code together in the standard way — we make our calculus class-based. Using a class-based calculus guarantees syntactic and type conformance to the interface, and the relationship between an interface and the objects which will implement it is much tighter in a class-based system.

Similar to the CORBA concept of *activation*, upon invocation the code gets copied from the body of its class and put in an agent which performs the work. The calling agent stalls until the method it invoked replies with the return value. It also supports the most general activation policy CORBA specifies for objects: Multiple calls on the same object, even the same method, can be serviced at once. Other policies can be implemented using a locking approach based on π-calculus channels.

2.2 Syntax

The bodies of Oompa methods contain *code*, generated by a syntax for process expressions similar to the polyadic π-calculus, extended with object primi-

tives and a typecase statement.

p	$::=$	$\texttt{fork}\{p_0\}\ p_1$	fork
		$\texttt{new}\ c\colon \mathrm{ChT}\ p_0$	create a new channel
		$c!\langle v_1, \ldots v_n\rangle\ p_0$	send
		$c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_0$	receive
		$o.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_m)\ p_0$	invoke a method
		$\texttt{create}\ o\colon C\ p_0$	create an object
		$a?r\ p_0$	access an attribute
		$a!v\ p_0$	update an attribute
		$\texttt{typecase}\ v\colon T\ \{p_0\}\ \texttt{else}\ \{p_1\}$	view v as a T
		$\texttt{end}$	stop

We will usually write code of the form $q\ \texttt{end}$ as just q. Code is a static description of behaviour, and is stored in a *method definition*, access to which is via the method's *signature*:

$$\mathrm{mdef} \ ::= \ \mathrm{sig}\{p\}$$

$$\mathrm{sig} \ ::= \ m?(r_1\colon T_1, \ldots r_n\colon T_n)!\langle d_1\colon T'_1, \ldots d_m\colon T'_{n'}\rangle$$

Classes contain a list of attributes whose declarations are given by the syntax:

$$\mathrm{adecl} \ ::= \ a\colon \texttt{Attr}\{T\}$$

So interfaces are a list of signatures and classes are a list of attributes declarations and method definitions:

$$\mathrm{Idef} \ ::= \ \texttt{interface}\ I\ \{\mathrm{sig}^*\}$$

$$\mathrm{Cdef} \ ::= \ \texttt{class}\ C\ \{\mathrm{adecl}^*\ \mathrm{mdef}^*\}$$

The definitions of interfaces and classes are collected into the *definition set*:

$$\Gamma \ ::= \ (\mathrm{Idef} \mid \mathrm{Cdef})^*$$

Upon method invocation, code is copied into an *agent* where the static descriptions of behaviour in the code are interpreted by the Oompa system as instructions.

g	$::=$	$\texttt{nil}$	no behaviour
		$o[p]$	executing code p from object o
		$(g_1 \mid g_2)$	parallel execution

The agent is labelled with the name of the object to which it belongs, which permits it access to the object's attributes.

Agents exist in a context given by a *type dictionary*, Φ, whose syntax is given by

$$\Phi ::= \{\text{tAss}^*\} \qquad \text{tAss} ::= v{:}T$$

and a *state dictionary*, Δ, whose syntax is given by

$$\Delta ::= \{\text{oAss}^*\} \qquad \text{oAss} ::= o := \{\text{aAss}^*\} \qquad \text{aAss} ::= a \mapsto v$$

An *Oompa configuration* is the triple consisting of an agent expression, g, a type dictionary, Φ, and a state dictionary, Δ. We write it $g\{^{\Phi}_{\Delta}$. Naturally, an Oompa configuration depends on the definition set, so an *Oompa system* is the quadruple $\Gamma \rhd g\{^{\Phi}_{\Delta}$.

2.3 Structural equivalence

The following seven rules of structural equivalence mean that "$\equiv$" is in fact an equivalence relation, and that "$|$" forms an Abelian monoid.

$$g \equiv g \qquad \frac{g_1 \equiv g_2 \qquad g_2 \equiv g_3}{g_1 \equiv g_3} \qquad \frac{g_1 \equiv g_2}{g_2 \equiv g_1} \qquad \frac{g_1 \equiv g_2}{(g_1 \mid g) \equiv (g_2 \mid g)}$$

$$(g_1 \mid \texttt{nil}) \equiv g_1 \qquad ((g_1 \mid g_2) \mid g_3) \equiv (g_1 \mid (g_2 \mid g_3)) \qquad (g_1 \mid g_2) \equiv (g_2 \mid g_1)$$

They essentially define a proof system that establishes the equivalence of agents which differ in the grouping and ordering of their subagents, and the presence of `nil`s.

2.4 Operational Semantics

We give here the rules of the *one-step operational semantics*, three of which are inference rules and the rest are axioms. The *statements* of the operational semantics are of the form $\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}$, i.e. a transition from one Oompa configuration to another in the context of a definition set Γ.

Of the inference rules, the first two are *Left* and *Right Equivalence* which entitle us swap structurally equivalent agent expressions. The third is *Parallelism* which allows us to talk about an agent's behaviour in a context containing other agents.

$$\frac{g \equiv g_1 \quad \Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}}{\Gamma \rhd g_1\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}} \qquad \frac{g' \equiv g_1 \quad \Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}}{\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g_1\{^{\Phi'}_{\Delta'}}$$

$$\frac{\Gamma \rhd g_1\{^{\Phi}_{\Delta} \longrightarrow g_1'\{^{\Phi'}_{\Delta'}}{\Gamma \rhd (g_1 \mid g_2)\{^{\Phi}_{\Delta} \longrightarrow (g_1' \mid g_2)\{^{\Phi'}_{\Delta'}}$$

The *End* axiom allows us to discard agents which have finished their work (since `end` concludes any piece of code).

$$\Gamma \ \rhd \ o[\texttt{end}]\{^{\Phi}_{\Delta} \ \longrightarrow \ \texttt{nil}\{^{\Phi}_{\Delta}$$

The *Fork* axiom allows an agent to create a new agent from the same object.

$$\Gamma \ \rhd \ o[\texttt{fork}\{p_1\}\ p_2]\{^{\Phi}_{\Delta} \ \longrightarrow \ o[p_1] \mid o[p_2]\{^{\Phi}_{\Delta}$$

The *New Channel* axiom allow an agent to create a new channel of a given channel type. Unlike the π-calculus, this doesn't operate as an restriction, but involves the choosing of a completely new name for the channel, and then its substitution into the code of the agent.

$$\Gamma \ \rhd \ o[\texttt{new}\ c\colon \mathrm{ChT}\ p]\{^{\Phi}_{\Delta} \ \longrightarrow \ o[p\{|^{c'}/_{c}|\}]\{^{\Phi\cup\{c':\mathrm{ChT}\}}_{\Delta}$$

where c' is a new name.

The *Communication* axiom is a typed version of the π-calculus one. The types do not affect the operation of the axiom, which substitutes the values as long as the sequences have the same length, but are used for static type checking.

$$\begin{array}{lll} \Gamma \quad \rhd & o_1[c!\langle v_1, \ldots v_n\rangle\ p_1] \ \mid & \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right. \\ & o_2[c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_2] & \\ \\ \longrightarrow & o_1[p_1] \ \mid & \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right. \\ & o_2[p_2\{|^{v_1}/_{r_1}, \ldots {}^{v_n}/_{r_n}|\}] & \end{array}$$

The *Method Invocation* axiom involves several steps. First the class of the target object is looked up in the type dictionary, Φ. If it has a method of the correct name, with input and output arities matching the call, then the code of the method is copied into a new agent belonging to the target object, and a new return channel is created. The input parameters and the special values `this` and `return` are substituted for their actual values in the new agent. The calling agent is blocked behind a receive on the return channel.

$$\begin{array}{lll} \Gamma \quad \rhd & o[o_1.m!\langle v_1, \ldots v_n\rangle?(s_1, \ldots s_l)\ p] & \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right. \\ \\ \longrightarrow & o[r?(s_1\colon T_1, \ldots s_l\colon T_l)\ p] \ \mid & \left\{ \begin{array}{l} \Phi' \\ \Delta \end{array} \right. \\ & o_1[p_1\{|^{v_1}/_{r_1}, \ldots {}^{v_n}/_{r_n}, {}^{o_1}/_{\texttt{this}}, {}^{r}/_{\texttt{return}}|\}] & \end{array}$$

where r is a new name, o_1's class in Φ has a method whose signature is $m?(r_1\colon S_1, \ldots r_n\colon S_n)!\langle d_1\colon T_1, \ldots d_l\colon T_l\rangle$, and $\Phi' = \Phi \cup \{r\colon \texttt{Chan}\langle T_1, \ldots T_l\rangle\}$.

The *Object Creation* axiom, like the New Channel axiom, involves the choosing of a completely new name, which is added to the type dictionary. The new object name is added to the state dictionary and substituted into the code of the agent.

$$\Gamma \ \triangleright \ o[\texttt{create}\ o_1{:}\,\mathrm{ClT}\ p]\{^{\Phi}_{\Delta} \ \longrightarrow \ o[p\{\!|^{o'_1}/_{o_1}|\!\}]\{^{\Phi\cup\{o'_1:\mathrm{ClT}\}}_{\Delta\cup\{o'_1=\emptyset\}}$$

where o'_1 is a new name.

The *Attribute Access* axiom applies when the object has a value assigned to the attribute. In the cases where the side condition fails, we can think of the non-application of this axiom as the agent blocking until an attribute value becomes available. If there is a value, it is substituted into the code of the agent in place of the formal parameter.

$$\Gamma \ \triangleright \ o[a?r\ p]\{^{\Phi}_{\Delta} \ \longrightarrow \ o[p\{\!|^{v}/_{r}|\!\}]\{^{\Phi}_{\Delta}$$

where $o \in \mathrm{Dom}(\Delta)$, $\Delta(o)(a)$ is defined and $\Delta(o)(a) = v$.

The *Attribute Update* axiom always applies, and overrides whatever value the state dictionary has for the attribute (even if it doesn't have one) with the given value.

$$\Gamma \ \triangleright \ o[a!v\ p]\{^{\Phi}_{\Delta} \ \longrightarrow \ o[p]\{^{\Phi}_{\Delta'}$$

where $o \in \mathrm{Dom}(\Delta)$, and $\Delta'(o_1)(a_1) = \Delta(o_1)(a_1)$ if $o_1 \neq o$ or $a_1 \neq a$ and v otherwise.

The *Typecase* axiom allows an agent to try to *view* a value as having a different type. This is possible as all values are marked in the type dictionary with their type and feasible as the subtyping system is terminating. Thus the question of whether this *assignment attempt* succeeds can be resolved in finite time.

$$\Gamma \ \triangleright \ o[\texttt{typecase}\ v{:}\,T\ \{p_0\}\ \texttt{else}\ \{p_1\}]\{^{\Phi}_{\Delta} \ \longrightarrow \ o[p_i]\{^{\Phi}_{\Delta}$$

where $i = 0$ if $\Gamma, \Phi \ \vdash \ v{:}\,T$ and 1 otherwise.

2.4.1 Initial System Convention

In order to populate our system with agents, we introduce a convention that allows us to mark a number of methods to be invoked at "system start". Say $\{C_1, \ldots C_n\}$ are those classes in the definition set, Γ, which have a method with signature main?()!$\langle\rangle$ and no attributes. Say that their bodies are $p_1, \ldots p_n$ and no p_i uses either `this` or `return`. Say Φ_0 is a type dictionary with no object types, and dummy_i are new object names. Then

$$\Gamma \ \triangleright \ \mathrm{dummy}_1[p_1] \mid \ldots \mid \mathrm{dummy}_n[p_n]\{^{\Phi_0}_{\emptyset}$$

is an *initial system*. We will sometimes use the notation g_Γ to denote the agent of this Oompa system.

2.4.2 The "$\Longrightarrow$" Relation

We generalise the one-step system to its reflexive transitive closure "$\Longrightarrow$", which allows us to reason about sequences of behaviour. This is generated by the inference system with the following rules:

$$\Gamma \rhd g\{^{\Phi}_{\Delta} \implies g\{^{\Phi}_{\Delta}$$

$$\frac{\Gamma \rhd g_1\{^{\Phi_1}_{\Delta_1} \implies g_2\{^{\Phi_2}_{\Delta_2} \qquad \Gamma \rhd g_2\{^{\Phi_2}_{\Delta_2} \longrightarrow g_3\{^{\Phi_3}_{\Delta_3}}{\Gamma \rhd g_1\{^{\Phi_1}_{\Delta_1} \implies g_3\{^{\Phi_3}_{\Delta_3}}$$

3 Types and Type Safety

The operational semantics are defined in terms of pattern matching, and make no use of the type decorations in the Oompa syntax. Consequently, they will permit behaviours we never intended and do not want. These behaviours fall into two main groups.

An example of the first is when an agent performs an invocation on an object which doesn't possess a matching method. This "stalls the system" in the sense that no rules of the operational semantics will apply. This is an example of the "round hole/square peg" class of type error — a square peg will not fit in a round hole.

An example of the second is when an agent assigns a value interpreted as an address to an attribute intended for first names. This involves the propagation of bad data, and is at least as serious as the first type. We might consider this the "round hole/finger" class of type error — a finger may actually fit in a hole, but it may then get stuck or electrocuted.

We deal with most of these *mis*behaviours using a form of type assignment system. We introduce types which describe the values a context can accept, and the contexts a value can be used in. We decorate the underlying syntax with these types[1], expressing our intended uses of the various elements of our system. In terms of these types, we formalise a notion of type violation, which describes the kind of misbehaviour we have been discussing.

Next, we provide a type safety system which tests an Oompa expression to see if it will commit a type violation. This last step is a form of *type assignment* where Oompa expressions which pass the test can be thought to be assigned the type "okay". Lastly, we exclude from study those expressions which fail this test. Currently, the efficiency of the algorithms and inference systems discussed in this technical report are not a primary concern.

[1] As we do not currently offer a type inference algorithm, these types must always be present.

3.1 Syntax of Types

The types are generated by the following syntax. First we have *Primitive Types* for which we list here `Long` and `Char`. For the present work, we consider these as sets of atomic constants, e.g. 10, 'a', etc.

$$\mathrm{PrT} \ ::= \ \texttt{Long} \mid \texttt{Char}$$

Next we have *Channel Types*, which are typed according to the arity of the tuples they can carry, the types of the values and the direction in which they can be used. While a channel itself is always bidirectional, a user of the channel may only have read or write access to it.

$$\mathrm{ChT} ::= \ \texttt{Chan}\langle T_1, \ldots T_n\rangle \mid \texttt{InCh}\langle T_1, \ldots T_n\rangle \mid \texttt{OuCh}\langle T_1, \ldots T_n\rangle$$

Interface types type objects according to the signature types of the methods that the objects support. *Signature types* depend on the type of values accepted and returned by the method, and the methods name.

$$\mathrm{InT} ::= \ \texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\} \qquad \mathrm{SgT} ::= \ m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_m\rangle$$

Guarded types are non-primitive types which have an outermost type-constructor. In our system they are either interface types or channel types. We introduce explicit recursive types using the `rec` operator, and restrict recursion to guarded types. *Value types*, the types of entities that can act as values, can therefore be primitive types, guarded types, type variables or recursive types.

$$T ::= \ \mathrm{PrT} \mid \mathrm{GrT} \mid t \mid \texttt{rec}\ t.\mathrm{GrT} \qquad\qquad \mathrm{GrT} ::= \ \mathrm{InT} \mid \mathrm{ChT}$$

Note that in $\texttt{rec}\ t.T$ occurrences of t are bound in T. α-substitution of bound variable names must be used to avoid prevent capture when substituting into a recursive type of this form.

The two non-value types in the system are signature types and *attribute types*, which label attributes with the type of values they can hold.

$$\mathrm{AtT} ::= \ \texttt{Attr}\{T\}$$

The presence of the type constructor $\texttt{Attr}\{\cdot\}$ prevents the system from confusing attribute names and the values they hold. For example, an attempt to send an attribute name, as in $c!\langle a\rangle$, will not type check.

3.2 The Typing System

The *typing system* is a proof system that establishes typing judgements of the form $\Gamma, \Phi \ \vdash \ v{:}T$ where Γ is a definition set, Φ is a type dictionary, v is some value or variable of the system and T is some type of the system. The statement can be read as "v is a T in the context Γ,Φ."

There are three rules of the typing system. The first two are *Introduction* and *Literal Introduction*.

$$\Gamma, \Phi \cup \{v{:}T\} \ \vdash \ v{:}T \qquad\qquad \Gamma, \Phi \ \vdash \ \ell_P{:}P$$

where v is a variable and ℓ_P is a literal of primitive type P. The third is *Subtyping* which uses the subtyping system defined in Section 5.

$$\frac{\Gamma \ \vdash \ T_1 \leq T_2 \qquad \Gamma, \Phi \ \vdash \ v{:}T_1}{\Gamma, \Phi \ \vdash \ v{:}T_2}$$

3.3 Type Violations

In terms of the types we have just defined, we formalise the notion of *type violation*. The idea is to decide the uses of a value which are not compatible with that value's type. Our choice of these determines the properties we expect of the type system.

Definition: In an Oompa system, $\Gamma \ \rhd \ g\{^{\Phi}_{\Delta}$, an agent $o[p] \in g$, is *type violating* if any of the following conditions hold:

- *Feature Mismatch:* The agent attempts to use a feature which doesn't exist by:
  - *invoking a method*, m, of an object, o'. $o'{:}C \in \Phi$, and p begins $o'.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_m)$, but either C is not a class or C's definition in Γ has no method m.
  - *accessing an attribute*, a, that o doesn't possess. So $o{:}C \in \Phi$, and p begins $a?r$, but C's definition in Γ has no attribute a.
  - *updating an attribute*, a, that o doesn't possess. So $o{:}C \in \Phi$, and p begins $a!v$, but C's definition in Γ has no attribute a.
- *Arity Mismatch:* The agent attempts to use a tuple of the wrong length by:
  - *receiving on a channel*, c. So $c{:}\texttt{Chan}\langle T_1, \ldots T_n\rangle \in \Phi$, but p begins $c?(r_1{:}T'_1, \ldots r_m{:}T'_m)$ where $n \neq m$.

- *sending on a channel*, c. So $c\colon \texttt{Chan}\langle T_1, \ldots T_n\rangle \in \Phi$, but p begins $c!\langle v_1, \ldots v_m\rangle$ where $n \neq m$.
- *invoking a method*, m, of an object, o'. So $o'\colon C \in \Phi$ and C's definition in Γ gives m the signature type $m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_m\rangle$. However, p begins $o'.m!\langle v_1, \ldots v_j\rangle?(r_1, \ldots r_k)$ where either $j \neq n$ or $k \neq m$.

- *Value Mismatch:* The agent attempts to use a value which is of the wrong type by:
  - *sending on a channel*, c. So $c\colon \texttt{Chan}\langle T_1, \ldots T_n\rangle \in \Phi$ and p begins $c!\langle v_1, \ldots v_n\rangle$ but for some i, we have $\Gamma, \Phi \nvdash v_i\colon T_i$.
  - *invoking a method*, m, of an object, o'. So $o'\colon C \in \Phi$, C's definition in Γ gives m the signature type $m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_m\rangle$, and p begins $o'.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_m)$. However, for some i, we have $\Gamma, \Phi \nvdash v_i\colon T_i$.
  - *updating an attribute*, a. So $o\colon C \in \Phi$, C's definition in Γ gives a the type $\texttt{Attr}\{T\}$, and p begins $a!v$. However, $\Gamma, \Phi \nvdash v\colon T$.

A system, $\Gamma \rhd g\{^{\Phi}_{\Delta}$, is *type violating* if any agent in g is type violating. It is useful to note that type violations are only associated with channel, method or attribute use.

Another situation we must beware of is where an agent attempts to create an object of a type that is not a class. This is certainly an error, but we choose not to call this a type violation as it does not actually represent the *misuse* of a value. We enable our type system to detect these errors at the cause, even though it would detect any invocation on such an object as a type violation.

3.4 Type Safety

The *Type Safety System* is a proof system that establishes type safety judgements of the form $\Gamma, \Phi \vdash x$ where Γ is the class library, Φ is a type dictionary and x is some Oompa expression. We are saying that x is well-behaved in the context of Γ and Φ. An alternative approach (used, for example, in [PS93]) gives an *okay type* to all expressions which are well-behaved in this way, e.g. $\Gamma, \Phi \vdash x\colon \mathit{okay}$ or $\Gamma, \Phi \vdash x\colon \circ$. The difference is essentially notational, and we prefer the more concise form as used, for example, in [San96].

3.4.1 Type Safety of Code

We start with rules for establishing the type-safety of a piece of code. There are ten rules, each rule being concerned with a single code primitive, a fact that we use later on to prove results about the type safety system. The first four rules deal with ends, forking, and channel and object creation.

$$\Gamma, \Phi \vdash \texttt{end} \qquad \frac{\Gamma, \Phi \vdash p_0 \qquad \Gamma, \Phi \vdash p_1}{\Gamma, \Phi \vdash \texttt{fork}\{p_0\}\ p_1}$$

$$\frac{\Gamma, \Phi \cup \{c\colon \mathrm{ChT}\} \vdash p}{\Gamma, \Phi \vdash \texttt{new}\ c\colon \mathrm{ChT}\ p} \qquad \frac{\Gamma, \Phi \cup \{o\colon C\} \vdash p}{\Gamma, \Phi \vdash \texttt{create}\ o\colon C\ p}$$

where the fourth rule requires that C is defined as a class in Γ.

The rules for sending and receiving with channels are straightforward.

$$\frac{\Gamma, \Phi \vdash c\colon \texttt{OuCh}\langle T_1, \ldots T_n\rangle \quad \Gamma, \Phi \vdash v_1\colon T_1 \cdots \Gamma, \Phi \vdash v_n\colon T_n \quad \Gamma, \Phi \vdash p}{\Gamma, \Phi \vdash c!\langle v_1, \ldots v_n\rangle\ p}$$

$$\frac{\Gamma, \Phi \vdash c\colon \texttt{InCh}\langle T_1, \ldots T_n\rangle \qquad \Gamma, \Phi \cup \{r_1\colon T_1, \ldots r_n\colon T_n\} \vdash p}{\Gamma, \Phi \vdash c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p}$$

The rule for checking method invocation requires the introduction of a new channel variable r. This separates the invocation from the return of values, as in the operational semantics, and facilitates convenient proofs later on.

$$\frac{\left(\begin{array}{c} \Gamma, \Phi \vdash o\colon \texttt{Intf}\{m?(T_1, \ldots T_n)!\langle T'_1 \ldots T'_{n'}\rangle\} \\ \Gamma, \Phi \vdash v_1\colon T_1 \qquad \cdots \qquad \Gamma, \Phi \vdash v_n\colon T_n \\ \Gamma, \Phi \cup \{r\colon \texttt{Chan}\langle T'_1, \ldots T'_{n'}\rangle\} \vdash r?(r_1\colon T'_1, \ldots r_{n'}\colon T'_{n'})p \end{array}\right)}{\Gamma, \Phi \vdash o.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_{n'})\ p}$$

where r is not in Φ.

The attribute access and update rules are straightforward.

$$\frac{\Gamma, \Phi \vdash a\colon \texttt{Attr}\{T\} \qquad \Gamma, \Phi \cup \{r\colon T\} \vdash p}{\Gamma, \Phi \vdash a?r\ p}$$

$$\frac{\Gamma, \Phi \vdash a\colon \texttt{Attr}\{T\} \qquad \Gamma, \Phi \vdash v\colon T \qquad \Gamma, \Phi \vdash p}{\Gamma, \Phi \vdash a!v\ p}$$

The rule for typecase requires both continuations to be type-safe, where $\Phi \dagger \{v\colon T\}$ means that $v\colon T'$ is replaced in Φ by $v\colon T$.

$$\frac{\Gamma, \Phi \dagger \{v\colon T\} \vdash p_0 \qquad \Gamma, \Phi \vdash p_1}{\Gamma, \Phi \vdash \texttt{typecase}\ v\colon T\ \{p_0\}\ \texttt{else}\ \{p_1\}}$$

where $v\colon T' \in \Phi$ for some T'.

3.4.2 Type Safety of Definitions

A method definition is type safe if its code will be type safe in the context of an invocation, e.g. when there are given input values, and a return channel.

$$\frac{\Gamma, \Phi \cup \{r_1\!:T_1, \ldots r_n\!:T_n, \texttt{return}\!:\texttt{OuCh}\langle T'_1, \ldots T'_{n'}\rangle\} \ \vdash p}{\Gamma, \Phi \ \vdash m?(r_1\!:T_1, \ldots r_n\!:T_n)!\langle d_1\!:T'_1, \ldots d_m\!:T'_{n'}\rangle\{p\}}$$

Type checking a class entails checking the method definitions in the context of that class, i.e. with the attributes and special value `this` of the appropriate type.

$$\frac{\left(\begin{array}{c}\Gamma, \Phi \cup \{a_1\!:\mathrm{AtT}_1, \ldots a_n\!:\mathrm{AtT}_n, \texttt{this}\!:C\} \ \vdash \mathrm{mdef}_1 \\ \vdots \\ \Gamma, \Phi \cup \{a_1\!:\mathrm{AtT}_1, \ldots a_n\!:\mathrm{AtT}_n, \texttt{this}\!:C\} \ \vdash \mathrm{mdef}_m\end{array}\right)}{\Gamma, \Phi \ \vdash \texttt{class}\ C\ \{\ a_1\!:\mathrm{AtT}_1\ \ldots\ a_n\!:\mathrm{AtT}_n\ \ \mathrm{mdef}_1\ \ldots\ \mathrm{mdef}_m\ \}}$$

We can now type check the definition set in a context consisting of some type dictionary Φ. We need to check that all the class definitions in the system are type safe with respect to each other.

$$\frac{\Gamma, \Phi \ \vdash \mathrm{Cdef}_1 \qquad \cdots \qquad \Gamma, \Phi \ \vdash \mathrm{Cdef}_n}{\Phi \ \vdash \Gamma}$$

where $\mathrm{Cdef}_1 \ldots \mathrm{Cdef}_n$ are all the class definitions in Γ.

4 Oompa Type Trees

Although we introduced a recursive type form into our syntax of types, we have not motivated it yet. There are two different ways in which a non-recursive type system would be insufficient for our purposes. Firstly, without recursive types our channels would be limited to carrying "simpler" values. This severely limits a system's dynamic behaviour, as the most complex channel types in the system would have to form a static infrastructure — there are no channels which could be used to transmit their names.

Second, in a object-oriented system like Oompa, interface types will often be found to be inter-referential. For example, the following two interfaces have this inter-dependency:

```
interface A
{
  m?()!<d:B>
}
```

```
interface B
{
  m?()!<d:A>
}
```

Consequently, we acknowledge these two kinds of recursion in our type system. The first kind we consider to be *explicit recursion* and consists of the use of the `rec` operator. For example, the type $\texttt{rec}\ t.\texttt{Chan}\langle\texttt{Long}, t\rangle$, which advertises its recursivity. The second kind is a consequence of using a class or interface name in a type position. Where we have a class or interface definition which uses other class or interface names (possibly including its own) we will must allow for the possibility of inter-dependency. Where there is this inter-dependency, we will consider it to be *implicit recursion*.

Once we allow these recurring structures into the system, we must be concerned for whether our type system meaningfully describes these objects. Therefore, we make the following distinction: The mathematical objects which encode the dependencies of the type system are (possibly) infinite trees. We will consider a syntactic type to be a description of a tree, and consider the set of Oompa type trees to be a model of the syntax of types.

Of particular use is the expansion function, defined in Section 4.2. This allows us to replace implicit recursion in types with explicit recursion using the `rec` operator, and removes the dependency of syntactic types on the definition set. We verify that this syntactic transformation preserves the interpretation in the tree semantics.

4.1 Building Trees from Types

The nodes on an Oompa type tree are labelled with symbols from the following *ranked alphabet*. The superscript of the symbols denotes the number of children the node it labels should have, and the symbols corresponding to signature types carry a method name.

$$\begin{aligned} L \;=\; & \{\texttt{Long}^0, \texttt{Char}^0\} \\ & \cup \{\mathrm{Sig}^2(m) \mid m \in \mathit{Id}\} \\ & \cup \{\mathrm{Intf}^n, \mathrm{Chan}^n, \mathrm{InCh}^n, \mathrm{OuCh}^n, \mathrm{InParam}^n, \mathrm{OutParam}^n \mid n \in \mathbb{W}\} \end{aligned}$$

We use the symbol $\mathbb{W}$ for the whole numbers[2] (i.e. $\mathbb{W} = \{0\} \cup \mathbb{N}$) and we will use l^n to stand for a general member of L.

We can descend a tree from its root by specifying, at each stage, the child we next want to visit as "the ith child from the left". Therefore, we can view a sequence of natural numbers as a path through a tree and represent trees as partial functions from paths to the label on the node reached by that path. Using the notation $\downarrow$ to mean "is defined", $\pi\sigma$ for the concatenation of π and σ and Λ for the empty sequence, we thus define an *Oompa type tree*, A, to be a partial function from $\mathbb{N}^*$ to L, which satisfies the following conditions:

[2]We take the view that the natural numbers are the counting numbers $1, 2, \ldots$

- $A(\Lambda)\downarrow$
- $A(\pi\sigma)\downarrow \implies A(\pi)\downarrow$
- $A(\pi) = l^n \implies A(\pi j)\downarrow$ where $j \leq n$
- $A(\pi) = \text{Intf}^n \implies \begin{array}{l} A(\pi j) = \text{Sig}^2(m) \text{ where } j \leq n \text{ and} \\ \text{if } A(\pi i) = \text{Sig}^2(m') \text{ for } i \neq j \text{ then } m \neq m' \end{array}$
- $A(\pi j) = \text{Sig}^2(m) \implies A(\pi) = \text{Intf}^n, \text{ some } n \in \mathbb{W} \text{ such that } n \geq j$
- $A(\pi) = \text{Sig}^2(m) \implies \begin{array}{l} A(\pi 1) = \text{InParam}^n, \text{ some } n \in \mathbb{W} \text{ and} \\ A(\pi 2) = \text{OutParam}^{n'}, \text{ some } n' \in \mathbb{W} \end{array}$
- $A(\pi j) = \text{InParam}^n \implies A(\pi) = \text{Sig}^2(m), \text{ some } m$
- $A(\pi j) = \text{OutParam}^n \implies A(\pi) = \text{Sig}^2(m), \text{ some } m$

Notice that type trees do not use type variables, so they are independent of the definition set. Thus, when we build the type tree of a type, we will recursively *look-up* the corresponding definition for each type variable and *expand* it into the tree.

We define the *look-up function* $\text{e}_\Gamma(\cdot)$ to perform this operation of looking-up and expanding definitions as:

$$\text{e}_\Gamma(t) = \texttt{Intf}\{\text{sig}_1^-, \ldots \text{sig}_n^-\}$$

if Γ contains either an interface of the form

$$\texttt{interface}\ t\ \{\text{sig}_1 \ldots \text{sig}_n\}$$

or a class of the form

$$\texttt{class}\ t\ \{\text{adecl}^*\ \text{sig}_1\{p_1\} \ldots \text{sig}_n\{p_n\}\}$$

where sig^- is the signature sig with all its parameter names removed. This simple syntactic step gives the signature type of the signature.

We now define the recursive function, Tree_Γ, which takes an Oompa type in the context of the definition set Γ and gives its corresponding Oompa type tree. The free type variables in the type are completely expanded using the

$\mathrm{e}_\Gamma(\cdot)$ function repeatedly.

$$
\begin{aligned}
&\mathrm{Tree}_\Gamma(\texttt{Long})(\Lambda) = \texttt{Long}^0\\
&\mathrm{Tree}_\Gamma(\texttt{Char})(\Lambda) = \texttt{Char}^0\\
&\mathrm{Tree}_\Gamma(\texttt{Chan}\langle T_1, \ldots T_n\rangle)(\Lambda) = \mathrm{Chan}^n\\
&\mathrm{Tree}_\Gamma(\texttt{Chan}\langle T_1, \ldots T_n\rangle)(i\pi) = \mathrm{Tree}_\Gamma(T_i)(\pi)\\
&\mathrm{Tree}_\Gamma(\texttt{InCh}\langle T_1, \ldots T_n\rangle)(\Lambda) = \mathrm{InCh}^n\\
&\mathrm{Tree}_\Gamma(\texttt{InCh}\langle T_1, \ldots T_n\rangle)(i\pi) = \mathrm{Tree}_\Gamma(T_i)(\pi)\\
&\mathrm{Tree}_\Gamma(\texttt{OuCh}\langle T_1, \ldots T_n\rangle)(\Lambda) = \mathrm{OuCh}^n\\
&\mathrm{Tree}_\Gamma(\texttt{OuCh}\langle T_1, \ldots T_n\rangle)(i\pi) = \mathrm{Tree}_\Gamma(T_i)(\pi)\\
&\mathrm{Tree}_\Gamma(\texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\})(\Lambda) = \mathrm{Intf}^n\\
&\mathrm{Tree}_\Gamma(\texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\})(i\pi) = \mathrm{Tree}_\Gamma(\mathrm{SgT}_i)(\pi)\\
&\mathrm{Tree}_\Gamma(m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle)(\Lambda) = \mathrm{Sig}^2(m)\\
&\mathrm{Tree}_\Gamma(m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle)(1) = \mathrm{InParam}^n\\
&\mathrm{Tree}_\Gamma(m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle)(2) = \mathrm{OutParam}^n\\
&\mathrm{Tree}_\Gamma(m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle)(1i\pi) = \mathrm{Tree}_\Gamma(T_i)(\pi)\\
&\mathrm{Tree}_\Gamma(m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle)(2i\pi) = \mathrm{Tree}_\Gamma(T'_i)(\pi)\\
&\mathrm{Tree}_\Gamma(\texttt{rec}\ t.T)(\pi) = \mathrm{Tree}_\Gamma(T\{\!|^{\texttt{rec}\ t.T}/_t|\!\})(\pi)\\
&\mathrm{Tree}_\Gamma(t)(\pi) = \mathrm{Tree}_\Gamma(\mathrm{e}_\Gamma(t))(\pi)
\end{aligned}
$$

The function is undefined in all other cases. Substitution on types is defined in Appendix A.

If our trees capture the relationships in the type system then we can now interpret our syntactic types as representing trees and hence encoding the necessary information.

4.2 Expanding Definitions

We introduce the *expansion function*, $\mathrm{E}^\emptyset_\Gamma(\cdot)$, that completely expands an Oompa type so that it no longer depends on the definition set. It does this by converting *implicit recursion*, which is due to recursive look-ups, into *explicit recursion*, which uses the `rec` operator. $\mathrm{E}^\emptyset_\Gamma(\cdot)$ is just a specific case of the recursive function $\mathrm{E}^V_\Gamma(\cdot)$ which looks-up all the variables it encounters

other that those in V.

$$
\begin{array}{l}
\mathrm{E}^V_\Gamma(\mathtt{Long}) = \mathtt{Long} \\
\mathrm{E}^V_\Gamma(\mathtt{Char}) = \mathtt{Char} \\
\mathrm{E}^V_\Gamma(\mathtt{Chan}\langle T_1, \ldots T_n\rangle) = \mathtt{Chan}\langle \mathrm{E}^V_\Gamma(T_1), \ldots \mathrm{E}^V_\Gamma(T_n)\rangle \\
\mathrm{E}^V_\Gamma(\mathtt{InCh}\langle T_1, \ldots T_n\rangle) = \mathtt{InCh}\langle \mathrm{E}^V_\Gamma(T_1), \ldots \mathrm{E}^V_\Gamma(T_n)\rangle \\
\mathrm{E}^V_\Gamma(\mathtt{OuCh}\langle T_1, \ldots T_n\rangle) = \mathtt{OuCh}\langle \mathrm{E}^V_\Gamma(T_1), \ldots \mathrm{E}^V_\Gamma(T_n)\rangle \\
\mathrm{E}^V_\Gamma(\mathtt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\}) = \mathtt{Intf}\{\mathrm{E}^V_\Gamma(\mathrm{SgT}_1), \ldots \mathrm{E}^V_\Gamma(\mathrm{SgT}_n)\} \\
\mathrm{E}^V_\Gamma(m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle) \\
\quad = m?(\mathrm{E}^V_\Gamma(T_1), \ldots \mathrm{E}^V_\Gamma(T_n))!\langle \mathrm{E}^V_\Gamma(T'_1), \ldots \mathrm{E}^V_\Gamma(T'_{n'})\rangle \\
\mathrm{E}^V_\Gamma(t) = \begin{cases} t & \text{if } t \in V \\ \mathtt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t)) & \text{otherwise} \end{cases} \\
\mathrm{E}^V_\Gamma(\mathtt{rec}\ t.T) = \mathtt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(T)
\end{array}
$$

In the last of these cases, we must be careful to avoid inappropriate variable capture. We use α-substitutability to rename t different to any other name in the system before applying this step of the expansion function.

We can view the definition set as a set of simultaneous equations, and the expansion function as an algorithm for solving them for a particular type variable. As an example, consider the interfaces defined at the beginning of this section. If we apply the expansion function to A we get:

$$\mathrm{E}^\emptyset_\Gamma(A) = \mathtt{rec}\ A.\mathtt{Intf}\{m?()!\langle \mathtt{rec}\ B.\mathtt{Intf}\{m?()!\langle A\rangle\}\rangle\}$$

Lemma 4.1 $\mathrm{E}^V_\Gamma(T)$ *is finite and can be calculated with finite applications of the look-up function* $\mathrm{e}_\Gamma(\cdot)$ *for any finite* Γ*,* T *and* V*.*

Proof: Consider the evaluation as a tree, where an application of the expansion function labels the nodes. Consider such a node $\mathrm{E}^{V'}_\Gamma(T')$. Let SubTerms[T] give the set of proper subterms of a type T [3]. We use the tuple $(\#V', \#\mathrm{SubTerms}[\mathrm{T}'])$ as a metric, with the following ordering:

$$(t_1, t_2) \leq (t'_1, t'_2) \text{ if } t_1 \geq t'_1 \text{ or } t_1 = t'_1 \text{ and } t_2 \leq t'_2$$

As we move down through the tree, either the first argument grows or it stays the same and the second shrinks. The first argument is bounded above by $\#V$ plus the number of occurrences of variables in Γ, which is certainly finite. The second is bounded below by 0. ■

We also define a recursive function which, for a given n, is like $\mathrm{Tree}_\Gamma(\cdot)$ except that it only performs n recursive unfoldings and definition look-ups.

[3]Not the same as the function Sub[·] defined in Section 5.3

We augment the language L with the symbol "$\perp^0$", and label nodes with this symbol at the point where an nth unfolding or look-up would previously occur. The different cases are as follows:

$$\begin{array}{ll} \mathrm{Tree}^0_\Gamma(\mathtt{rec}\ t.T)(\Lambda) = \perp^0 & \mathrm{Tree}^{n+1}_\Gamma(\mathtt{rec}\ t.T)(\pi) = \mathrm{Tree}^n_\Gamma(T\{\!|^{\mathtt{rec}\ t.T}/t|\!\}) \\ \mathrm{Tree}^0_\Gamma(t)(\Lambda) = \perp^0 & \mathrm{Tree}^{n+1}_\Gamma(t)(\pi) = \mathrm{Tree}^n_\Gamma(\mathrm{e}_\Gamma(t))(\pi) \end{array}$$

Lemma 4.2 $\mathrm{Tree}_\Gamma(T)(\pi) = \mathrm{Tree}^n_\Gamma(T)(\pi)$ *for all* $n > |\pi|$

Proof: We use induction on the length of π. The base cases are when $\pi = \Lambda$. Consider the structure of T. The base cases for the guarded types and the case for signature types are all the same, since $\mathrm{Tree}_\Gamma(T)$ and $\mathrm{Tree}^n_\Gamma(T)$ are the same on these types.

Say $T = t$, some variable. Then, since $n > 0$ we have:

$$\begin{array}{rcl} \mathrm{Tree}_\Gamma(t)(\Lambda) & = & \mathrm{Tree}_\Gamma(\mathrm{e}_\Gamma(t))(\Lambda) \\ \mathrm{Tree}^n_\Gamma(t)(\Lambda) & = & \mathrm{Tree}^{n-1}_\Gamma(\mathrm{e}_\Gamma(t))(\Lambda) \end{array}$$

In this case both will equal to Intf^m, as that will be the outermost type constructor of $\mathrm{e}_\Gamma(t)$.

Say $T = \mathtt{rec}\ t.T'$. Then T' is guarded and since $n > 0$ we have:

$$\begin{array}{rcl} \mathrm{Tree}_\Gamma(\mathtt{rec}\ t.T')(\Lambda) & = & \mathrm{Tree}_\Gamma(T'\{\!|^{\mathtt{rec}\ t.T'}/t|\!\})(\Lambda) \\ \mathrm{Tree}^n_\Gamma(\mathtt{rec}\ t.T')(\Lambda) & = & \mathrm{Tree}^{n-1}_\Gamma(T'\{\!|^{\mathtt{rec}\ t.T'}/t|\!\})(\Lambda) \end{array}$$

These will be the same, as the two functions are the same on guarded types, giving us the result in this case.

The inductive cases occur when $\pi = i\pi'$. Again we consider the structure of T.

Say $T = t$, some variable. Then if $\mathrm{e}_\Gamma(t)$ has signature types $\mathrm{SgT}_1, \ldots \mathrm{SgT}_k$ where $k \geq i$ we have

$$\mathrm{Tree}_\Gamma(t)(\pi) = \mathrm{Tree}_\Gamma(\mathrm{e}_\Gamma(t))(i\pi') = \mathrm{Tree}_\Gamma(\mathrm{SgT}_i)(\pi')$$

$$\mathrm{Tree}^n_\Gamma(t)(\pi) = \mathrm{Tree}^{n-1}_\Gamma(\mathrm{e}_\Gamma(t))(i\pi') = \mathrm{Tree}^{n-1}_\Gamma(\mathrm{SgT}_i)(\pi')$$

In this case they are equal by the inductive hypothesis. In the case where $\mathrm{e}_\Gamma(t)$ has less than i signatures both are undefined.

Say $T = \mathtt{rec}\ t.T'$ and T' has immediate subterms $T_1, \ldots T_k$ where $k \geq i$. Then T' is guarded and

$$\begin{array}{rcl} \mathrm{Tree}_\Gamma(\mathtt{rec}\ t.T')(\pi) & = & \mathrm{Tree}_\Gamma(T'\{\!|^{\mathtt{rec}\ t.T'}/t|\!\})(i\pi') \\ & = & \mathrm{Tree}_\Gamma(T_i\{\!|^{\mathtt{rec}\ t.T'}/t|\!\})(\pi') \end{array}$$

$$\begin{aligned}\mathrm{Tree}^n_\Gamma(\mathtt{rec}\ t.T')(\pi) &= \mathrm{Tree}^{n-1}_\Gamma(T'\{\!|\mathtt{rec}\ t.T'/t|\!\})(i\pi') \\ &= \mathrm{Tree}^{n-1}_\Gamma(T_i\{\!|\mathtt{rec}\ t.T'/t|\!\})(\pi')\end{aligned}$$

and these are equal by the inductive hypothesis. In the case where T' has less than i immediate subterms both are undefined.

The guarded cases and the case for signature types are all the same. Say T is some guarded type with immediate subterms $T_1, \ldots T_k$ where $k \geq i$. Then

$$\mathrm{Tree}_\Gamma(T)(i\pi') = \mathrm{Tree}_\Gamma(T_i)(\pi')$$
$$\mathrm{Tree}^n_\Gamma(T)(i\pi') = \mathrm{Tree}^{n-1}_\Gamma(T_i)(\pi')$$

These are the same by the induction hypothesis. In the case where T has less than i immediate subterms both are undefined. ■

We now introduce a notion of equivalence between two types, based on the equalities of their trees. We say $\Gamma \models A = B$ if and only if $\mathrm{Tree}_\Gamma(A) = \mathrm{Tree}_\Gamma(B)$. The next result validates the expansion function, by showing that the type it generates is equivalent to the first, in the context Γ.

Theorem 4.3 $\Gamma \models S = \mathrm{E}^\emptyset_\Gamma(S)$

Proof: We have to prove

$$\mathrm{Tree}_\Gamma(S) = \mathrm{Tree}_\Gamma(\mathrm{E}^\emptyset_\Gamma(S))$$

We prove this by showing that for any n and set of variables V,

$$\mathrm{Tree}^n_\Gamma(S) = \mathrm{Tree}^n_\Gamma(\mathrm{E}^V_\Gamma(S))$$

by induction on n. The theorem statement follows from Lemma 4.2, since for any argument sequence π we can pick $n = |\pi| + 1$ and $V = \emptyset$.

The base case is where $n = 0$. We use another induction on the construction of S. If $S = \mathtt{Long}$, or $S = \mathtt{Char}$, then $\mathrm{E}^V_\Gamma(S) = S$. The trees are obviously the same in these cases.

If $S = t$, there are two sub cases: Firstly, if $t \in V$ then $\mathrm{E}^V_\Gamma(S) = S$, and again the trees are the same. If $t \notin V$ then $\mathrm{E}^V_\Gamma(t) = \mathtt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))$. But then, given π, we have that $\mathrm{Tree}^0_\Gamma(t)(\pi)$ and $\mathrm{Tree}^0_\Gamma(\mathtt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t)))(\pi)$ are either both $\bot^0$ if $\pi = \Lambda$ or both undefined. Thus the trees are the same.

If $S = \mathtt{rec}\ t.T$, then $\mathrm{E}^V_\Gamma(S) = \mathtt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(T)$ and, given π, we have that $\mathrm{Tree}^0_\Gamma(\mathtt{rec}\ t.T)(\pi)$ and $\mathrm{Tree}^0_\Gamma(\mathtt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(T))(\pi)$ are either both $\bot^0$ if $\pi = \Lambda$ or both undefined. So the trees are the same.

If $S = \texttt{Chan}\langle T_1, \ldots T_n\rangle$, then $\mathrm{E}^V_\Gamma(S) = \texttt{Chan}\langle \mathrm{E}^V_\Gamma(T_1) \ldots \mathrm{E}^V_\Gamma(T_n)\rangle$. The trees are the same since:

$$\begin{array}{rcl} \mathrm{Tree}^0_\Gamma(S)(\Lambda) & = & \mathrm{Chan}^n \\ \mathrm{Tree}^0_\Gamma(\mathrm{E}^V_\Gamma(S))(\Lambda) & = & \mathrm{Chan}^n \\ \mathrm{Tree}^0_\Gamma(S)(i\pi) & = & \mathrm{Tree}^0_\Gamma(T_i)(\pi) \\ \mathrm{Tree}^0_\Gamma(\mathrm{E}^V_\Gamma(S))(i\pi) & = & \mathrm{Tree}^0_\Gamma(\mathrm{E}^V_\Gamma(T_i))(\pi) \end{array}$$

The last two are equal by the local inductive hypothesis. The other inductive cases for S are like this channel case.

We now consider the cases when $n > 0$. Again, we use another induction on the construction of S. If $S = \texttt{Long}$, or $S = \texttt{Char}$ then $S = \mathrm{E}^V_\Gamma(S)$, so the trees must be the same. Note that we won't be able to use the local inductive hypothesis in the cases for $S = t$ and $S = \texttt{rec}\ t.T$. The main inductive hypothesis will be sufficient. If $S = t$ and $t \in V$ then since $\mathrm{E}^V_\Gamma(t) = t$, the trees are the same.

If $S = t$ where $t \notin V$, then

$$\mathrm{E}^V_\Gamma(S) = \mathrm{E}^V_\Gamma(t) = \texttt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))$$

and

$$\mathrm{Tree}^n_\Gamma(\texttt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))) = \mathrm{Tree}^{n-1}_\Gamma(\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))\{\!|\texttt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))/t|\!\})$$

Also

$$\begin{array}{rcl} \mathrm{Tree}^n_\Gamma(t) & = & \mathrm{Tree}^{n-1}_\Gamma(\mathrm{e}_\Gamma(t)) \\ & = & \mathrm{Tree}^{n-1}_\Gamma(\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))) \end{array}$$

by the induction hypothesis.

Therefore, it will be sufficient to show that the following two trees are equal:

$$\begin{array}{l} \mathrm{Tree}^{n-1}_\Gamma(\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))\{\!|\texttt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))/t|\!\}) \\ \mathrm{Tree}^{n-1}_\Gamma(\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))) \end{array}$$

It is easy to see that the only place they could differ is at the subtrees where $\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))$ has a free occurrence of t. For one such occurrence the two subtrees will have the form:

$$\begin{array}{l} \mathrm{Tree}^{n-1-m}_\Gamma(\texttt{rec}\ t.\mathrm{E}^{V\cup\{t\}}_\Gamma(\mathrm{e}_\Gamma(t))) \\ \mathrm{Tree}^{n-1-m}_\Gamma(t) \end{array}$$

for some $m \geq 0$ (m will be the number of $\texttt{rec}$ operators we have passed through). We rely on the convention that bound variables are renamed to

guarantee that none of the free variables of $\texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(\mathrm{e}_\Gamma(t))$ will be captured when it is substituted into $\mathrm{E}_\Gamma^{V\cup\{t\}}(\mathrm{e}_\Gamma(t))$. Thus, no substitution generated when the Tree function encounters the $\texttt{rec}$ operator will apply to it. This justifies the form of the first subtree above.

By the induction hypothesis applied to the second subtree:

$$\begin{aligned}\mathrm{Tree}_\Gamma^{n-1-m}(t) &= \mathrm{Tree}_\Gamma^{n-1-m}(\mathrm{E}_\Gamma^{V\cup\{t\}}(t)) \\ &= \mathrm{Tree}_\Gamma^{n-1-m}(\texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(\mathrm{e}_\Gamma(t)))\end{aligned}$$

Thus, these subtrees are the same. Thus the main trees must be equal.

Say $S = \texttt{rec}\ t.T$, and use the variable convention to make sure that t is chosen different to any other name in the system. Then $\mathrm{E}_\Gamma^V(S) = \texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(T)$ and

$$\begin{aligned}\mathrm{Tree}_\Gamma^n(\mathrm{E}_\Gamma^V(S)) &= \mathrm{Tree}_\Gamma^n(\texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(T)) \\ &= \mathrm{Tree}_\Gamma^{n-1}(\mathrm{E}_\Gamma^{V\cup\{t\}}(T)\{\!|^{\texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(T)}/_{t}|\!\})\end{aligned}$$

Also by the main inductive hypothesis,

$$\begin{aligned}\mathrm{Tree}_\Gamma^n(\texttt{rec}\ t.T) &= \mathrm{Tree}_\Gamma^{n-1}(T\{\!|^{\texttt{rec}\ t.T}/_{t}|\!\}) \\ &= \mathrm{Tree}_\Gamma^{n-1}(\mathrm{E}_\Gamma^{V\cup\{t\}}(T\{\!|^{\texttt{rec}\ t.T}/_{t}|\!\}))\end{aligned}$$

Therefore, it will be sufficient to show that the following two trees are equal:

$$\begin{array}{l}\mathrm{Tree}_\Gamma^{n-1}(\mathrm{E}_\Gamma^{V\cup\{t\}}(T)\{\!|^{\texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(T)}/_{t}|\!\}) \\ \mathrm{Tree}_\Gamma^{n-1}(\mathrm{E}_\Gamma^{V\cup\{t\}}(T\{\!|^{\texttt{rec}\ t.T}/_{t}|\!\}))\end{array}$$

These trees will have the same "upper structure", and they could only differ at the subtrees where T has free variables. There are two cases.

Consider an occurrence of t in T. The subtrees at this point will have the form

$$\begin{array}{l}\mathrm{Tree}_\Gamma^{n-1-m}(\texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(T)) \\ \mathrm{Tree}_\Gamma^{n-1-m}(\mathrm{E}_\Gamma^{V\cup U\cup\{t\}}(\texttt{rec}\ t.T))\end{array}$$

for some $m \geq 0$ and set of variables U. Again, we rely on the convention that bound variables are renamed to guarantee that none of the free variables of $\texttt{rec}\ t.T$ are captured when it is substituted into T. Thus, no substitution generated when the Tree function encounters the $\texttt{rec}$ operator will apply to it.

The first subtree is equal to

$$\mathrm{Tree}_\Gamma^{n-1-m}(\mathrm{E}_\Gamma^V(\texttt{rec}\ t.T))$$

and by the inductive hypothesis twice this is equal to

$$\mathrm{Tree}_\Gamma^{n-1-m}(\texttt{rec}\ t.T) = \mathrm{Tree}_\Gamma^{n-1-m}(\mathrm{E}_\Gamma^{V\cup U\cup\{t\}}(\texttt{rec}\ t.T))$$

Thus in this case, the subtrees are the same.

Next, consider an occurrence of s in T, where $s \neq t$. The subtrees at this point will be

$$\mathrm{Tree}_\Gamma^{n-1-m}(\mathrm{E}_\Gamma^{V\cup U\cup\{t\}}(s)\rho_1 \ldots \rho_k)$$
$$\mathrm{Tree}_\Gamma^{n-1-m}(\mathrm{E}_\Gamma^{V\cup U\cup\{t\}}(s)\{\!|\texttt{rec}\ t.\mathrm{E}_\Gamma^{V\cup\{t\}}(T)/t|\!\}\rho_1 \ldots \rho_k)$$

where $\rho_1 \ldots \rho_k$ are substitutions generated when the Tree function encounters the `rec` operator. As t was chosen different from any other name, the extra substitution in the second subtree can be dropped — it won't apply to anything. Thus, these subtrees are the same, and hence the main trees are the same.

Say $S = \texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_m\}$. Then

$$\mathrm{E}_\Gamma^V(S) = \texttt{Intf}\{\mathrm{E}_\Gamma^V(\mathrm{SgT}_1), \ldots \mathrm{E}_\Gamma^V(\mathrm{SgT}_m)\}$$

The trees are the same, since:

$$\begin{aligned}
\mathrm{Tree}_\Gamma^n(S)(\Lambda) &= \mathrm{Intf}^m \\
\mathrm{Tree}_\Gamma^n(\mathrm{E}_\Gamma^V(S))(\Lambda) &= \mathrm{Intf}^m \\
\mathrm{Tree}_\Gamma^n(S)(i\pi) &= \mathrm{Tree}_\Gamma^n(\mathrm{SgT}_i)(\pi) \\
\mathrm{Tree}_\Gamma^n(\mathrm{E}_\Gamma^V(S))(i\pi) &= \mathrm{Tree}_\Gamma^n(\mathrm{E}_\Gamma^V(\mathrm{SgT}_i))(\pi)
\end{aligned}$$

The last two are equal by the local inductive hypothesis. The other inductive cases for S are all like this one, so we're done. ■

5 The Subtyping System

We are using Oompa type trees to encode the dependencies in the type system. Therefore, it is appropriate to consider subtyping with respect to the trees. We introduce a relation between Oompa type trees that captures the notion of subtype. As the trees are possibly infinite structures, the relation is a simulation which compares the structure of trees in terms of the labelling and subtrees. We then introduce an algorithm which operates on types which is terminating. We prove that the algorithm is sound and complete with respect to the tree simulation.

5.1 Tree Simulation

A relation $\mathcal{R}$ between Oompa type trees is an *Oompa tree simulation* if $(A, B) \in \mathcal{R}$ implies:

- If $B(\Lambda) = \texttt{Long}^0$ then $A(\Lambda) = \texttt{Long}^0$.
- If $B(\Lambda) = \texttt{Char}^0$ then $A(\Lambda) = \texttt{Char}^0$.
- If $B(\Lambda) = \text{Chan}^n$ then $A(\Lambda) = \text{Chan}^n$ and for each $0 \leq i < n$ both $(A(i), B(i)) \in \mathcal{R}$ and $(B(i), A(i)) \in \mathcal{R}$.
- If $B(\Lambda) = \text{InCh}^n$ then $A(\Lambda) = \text{InCh}^n$ or Chan^n, and for each $0 \leq i < n$, $(A(i), B(i)) \in \mathcal{R}$.
- If $B(\Lambda) = \text{OuCh}^n$ then $A(\Lambda) = \text{OuCh}^n$ or Chan^n, and for each $0 \leq i < n$, $(B(i), A(i)) \in \mathcal{R}$.
- If $B(\Lambda) = \text{Intf}^n$ then $A(\Lambda) = \text{Intf}^m$ where $m \geq n$, and there exists an injective function $f : [1..n] \to [1..m]$ such that for each $0 \leq i < n, (A(f(i)), B(i)) \in \mathcal{R}$.
- If $B(\Lambda) = \text{Sig}^2(m)$ then $A(\Lambda) = \text{Sig}^2(m)$ and for each $0 \leq i < 2$, $(A(i), B(i)) \in \mathcal{R}$
- If $B(\Lambda) = \text{InParam}^n$ then $A(\Lambda) = \text{InParam}^n$ and for each $0 \leq i < n$, $(B(i), A(i)) \in \mathcal{R}$.
- If $B(\Lambda) = \text{OutParam}^n$ then $A(\Lambda) = \text{OutParam}^n$ and for each $0 \leq i < n$, $(A(i), B(i)) \in \mathcal{R}$.

We use the notation $A \leq_{\text{tr}} B$ to indicate that $(A, B) \in \mathcal{R}$ for some tree simulation $\mathcal{R}$. For two types S and T, we use the notation $\Gamma \models S \leq T$ if and only if $\text{Tree}_\Gamma(S) \leq_{\text{tr}} \text{Tree}_\Gamma(T)$. As usual, the intended interpretation of $\Gamma \models S \leq T$ is that an entity of type S will do when an entity of type T is required.

5.2 The Subtyping Algorithm

We now define the algorithm which establishes the subtype relationship between two closed types, i.e. types which contain no free type variable.

Definition: The *subtyping algorithm* is a proof system whose statements have the form $\Sigma \vdash_{\text{a}} S \leq T$, where S and T are closed types and Σ is a set of assumptions, which have the form $U_1 \leq U_2$. The proof system is considered

an algorithm by putting an order on the rules — when more than one are applicable, choose the first as they are presented.

The two axioms of the system are *Reflexivity* and *Assumption*:

$$\Sigma \ \vdash_{\mathrm{a}} \ T \leq T \qquad\qquad \Sigma \cup \{S \leq T\} \ \vdash_{\mathrm{a}} \ S \leq T$$

Next we have the rules for *Channel Subtyping*. We use the two-way judgements to indicate that judgements in both directions are premises.

$$\frac{\Sigma \ \vdash_{\mathrm{a}} \ S_1 \lesseqgtr T_1 \quad \cdots \quad \Sigma \ \vdash_{\mathrm{a}} \ S_n \lesseqgtr T_n}{\Sigma \ \vdash_{\mathrm{a}} \ \mathtt{Chan}\langle S_1, \ldots S_n\rangle \leq \mathtt{Chan}\langle T_1, \ldots T_n\rangle}$$

$$\frac{\Sigma \ \vdash_{\mathrm{a}} \ S_1 \leq T_1 \quad \cdots \quad \Sigma \ \vdash_{\mathrm{a}} \ S_n \leq T_n}{\Sigma \ \vdash_{\mathrm{a}} \ \mathtt{InCh}\langle S_1, \ldots S_n\rangle \leq \mathtt{InCh}\langle T_1, \ldots T_n\rangle}$$

$$\frac{\Sigma \ \vdash_{\mathrm{a}} \ S_1 \leq T_1 \quad \cdots \quad \Sigma \ \vdash_{\mathrm{a}} \ S_n \leq T_n}{\Sigma \ \vdash_{\mathrm{a}} \ \mathtt{Chan}\langle S_1, \ldots S_n\rangle \leq \mathtt{InCh}\langle T_1, \ldots T_n\rangle}$$

$$\frac{\Sigma \ \vdash_{\mathrm{a}} \ T_1 \leq S_1 \quad \cdots \quad \Sigma \ \vdash_{\mathrm{a}} \ T_n \leq S_n}{\Sigma \ \vdash_{\mathrm{a}} \ \mathtt{OuCh}\langle S_1, \ldots S_n\rangle \leq \mathtt{OuCh}\langle T_1, \ldots T_n\rangle}$$

$$\frac{\Sigma \ \vdash_{\mathrm{a}} \ T_1 \leq S_1 \quad \cdots \quad \Sigma \ \vdash_{\mathrm{a}} \ T_n \leq S_n}{\Sigma \ \vdash_{\mathrm{a}} \ \mathtt{Chan}\langle S_1, \ldots S_n\rangle \leq \mathtt{OuCh}\langle T_1, \ldots T_n\rangle}$$

The rule for *Signature Subtyping* is as follows:

$$\frac{\begin{array}{c}\Sigma \ \vdash_{\mathrm{a}} \ T_1 \leq S_1 \quad \cdots \quad \Sigma \ \vdash_{\mathrm{a}} \ T_n \leq S_n \\ \Sigma \ \vdash_{\mathrm{a}} \ S'_1 \leq T'_1 \quad \cdots \quad \Sigma \ \vdash_{\mathrm{a}} \ S'_{n'} \leq T'_{n'}\end{array}}{\Sigma \ \vdash_{\mathrm{a}} \ m?(S_1, \ldots S_n)!\langle S'_1, \ldots S'_{n'}\rangle \leq m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle}$$

The rule for *Interface Subtyping* is:

$$\frac{\Sigma \ \vdash_{\mathrm{a}} \ \mathrm{SgT}_{f(1)} \leq \mathrm{SgT}'_1 \quad \cdots \quad \Sigma \ \vdash_{\mathrm{a}} \ \mathrm{SgT}_{f(n)} \leq \mathrm{SgT}'_n}{\begin{array}{rl}\Sigma \ \vdash_{\mathrm{a}} & \mathtt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n, \mathrm{SgT}_{n+1}, \ldots \mathrm{SgT}_{n+m}\} \\ & \leq \ \mathtt{Intf}\{\mathrm{SgT}'_1, \ldots \mathrm{SgT}'_n\}\end{array}}$$

if there is such an injective function, f, from $[1..n]$ to $[1..(n+m)]$.

There are two rules for explicitly recursive types. If the premise can be proved with the antecedent as an assumption, then we conclude that the antecedent is true. The two rules are *Left Recursive Subtyping* and *Right Recursive Subtyping*.

$$\frac{\Sigma \cup \{\mathtt{rec}\ s.S \leq T\} \ \vdash_{\mathrm{a}} \ S\{\!|^{\mathtt{rec}\ s.S}/_{s}|\!\} \leq T}{\Sigma \ \vdash_{\mathrm{a}} \ \mathtt{rec}\ s.S \leq T}$$

$$\frac{\Sigma \cup \{S \leq \mathtt{rec}\ t.T\} \ \vdash_{\mathrm{a}} \ S \leq T\{\!|^{\mathtt{rec}\ t.T}/_{t}|\!\}}{\Sigma \ \vdash_{\mathrm{a}} \ S \leq \mathtt{rec}\ t.T}$$

We extend the subtyping algorithm to all Oompa types by using the expansion function to close types. We write $\Gamma \vdash S \leq T$ if and only if $\emptyset \vdash_{\mathrm{a}} \mathrm{E}^{\emptyset}_{\Gamma}(S) \leq \mathrm{E}^{\emptyset}_{\Gamma}(T)$ is provable from the subtyping algorithm.

The subtyping algorithm is called *terminating* if any application of the algorithm to a pair of types generates a finite proof tree. It is called *sound* if $\Gamma \vdash S \leq T$ implies $\Gamma \models S \leq T$, and *complete* if $\Gamma \models S \leq T$ implies $\Gamma \vdash S \leq T$. As we shall see, the subtyping algorithm has all these three properties. This means we can use the algorithm applied to two Oompa types to establish the simulation relationship between the Oompa type trees they represent.

5.3 Termination

We need the following technical construction to facilitate some of our proofs. We define a function Sub[·], which gives the set of subterms of a closed type including recursive unfoldings:

$$\begin{aligned}
\mathrm{Sub}[\texttt{Chan}\langle T_1, \ldots T_n\rangle] &= \{\texttt{Chan}\langle T_1, \ldots T_n\rangle\} \\
&\quad \cup \mathrm{Sub}[T_1] \cup \ldots \mathrm{Sub}[T_n] \\
\mathrm{Sub}[\texttt{InCh}\langle T_1, \ldots T_n\rangle] &= \{\texttt{InCh}\langle T_1, \ldots T_n\rangle\} \\
&\quad \cup \mathrm{Sub}[T_1] \cup \ldots \mathrm{Sub}[T_n] \\
\mathrm{Sub}[\texttt{OuCh}\langle T_1, \ldots T_n\rangle] &= \{\texttt{OuCh}\langle T_1, \ldots T_n\rangle\} \\
&\quad \cup \mathrm{Sub}[T_1] \cup \ldots \mathrm{Sub}[T_n] \\
\mathrm{Sub}[\texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\}] &= \{\texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\}\} \\
&\quad \cup \mathrm{Sub}[\mathrm{SgT}_1] \cup \ldots \mathrm{Sub}[\mathrm{SgT}_n] \\
\mathrm{Sub}[m?(S_1, \ldots S_n)!\langle T_1, \ldots T_{n'}\rangle] &= \{m?(S_1, \ldots S_n)!\langle T_1, \ldots T_{n'}\rangle\} \\
&\quad \cup \mathrm{Sub}[S_1] \cup \ldots \mathrm{Sub}[S_n] \\
&\quad \cup \mathrm{Sub}[T_1] \cup \ldots \mathrm{Sub}[T_{n'}] \\
\mathrm{Sub}[\texttt{rec}\ t.T] &= \{\texttt{rec}\ t.T\} \\
&\quad \cup \{S\{\!|^{\texttt{rec}\ t.T}/_{t}|\!\} \mid S \in \mathrm{Sub}[T]\}
\end{aligned}$$

We also use $\mathrm{Sub}[S, T]$ to represent $\mathrm{Sub}[S] \cup \mathrm{Sub}[T]$.

Lemma 5.1 *For a finite closed type T, $\mathrm{Sub}[T]$ is finite*

Proof: $\mathrm{Sub}[T]$ can have no more elements than the distinct subterms of T. ■

Theorem 5.2 ***(Termination)*** *The subtyping algorithm is terminating.*

Proof: Consider the algorithm applied to $\vdash_{\mathrm{a}} S \leq T$ where S and T are closed. Consider a node in the proof tree $\Sigma \vdash_{\mathrm{a}} S' \leq T'$. We claim that this node satisfies the following three properties:

1. S' and T' are in $\mathrm{Sub}[S,T]$.

2. For every assumption $U_1 \leq U_2 \in \Sigma$, both U_1 and U_2 are in $\mathrm{Sub}[S,T]$.

3. No assumption occurs in Σ more than once.

These properties clearly hold for the root node. We show that if the current goal node satisfies them, then so do the premises.

In the cases of Reflexivity and Assumption there are no premises. Consider the Chan-InCh rule. $S = \mathtt{Chan}\langle S_1, \ldots S_n\rangle \in \mathrm{Sub}[S,T]$ so by the definition of $\mathrm{Sub}[\cdot]$, $S_1, \ldots S_n \in \mathrm{Sub}[S,T]$, and $T = \mathtt{InCh}\langle T_1, \ldots T_n\rangle \in \mathrm{Sub}[S,T]$, so $T_1, \ldots T_n \in \mathrm{Sub}[S,T]$. This means that the first property holds for all the premises. This rule adds no new assumptions, so the second and third property hold. The cases for the other channel subtyping rules and the interface and signature subtyping rules are the same.

Consider the right-recursive subtyping rule. S' was in $\mathrm{Sub}[S,T]$ as it is one of the types in the antecedent. $T'\{\!|^{\mathtt{rec}\ t.T'}/_t|\!\}$ is in $\mathrm{Sub}[S,T]$, since $\mathtt{rec}\ t.T'$ is. Thus the first property holds. The second property comes from the fact that the added assumption is exactly the antecedent, so both it's elements are in $\mathrm{Sub}[S,T]$. The third property comes from the fact that if we added an assumption which was already there, it would have contradicted the algorithm order, which stipulates that the Assumption rule be attempted before the Right Recursive rule. The left recursive rule can be dealt with similarly.

We now associate a measure with each node. Let

$$M(\Sigma \ \vdash_{\mathrm{a}} \ S \leq T) = (n, m)$$

where n is the number of assumptions in Σ and m is the maximum nesting of brackets in either S or T. We say that $(n,m) > (n',m')$ if $n < n'$ or else $n = n'$ and $m > m'$.

It is easy to see that at any application of the rules, the measure of any of the premises is less than that of the antecedent. The assumption and reflexivity rules don't generate any new premises, the recursion rules add an assumption and the other rules all reduce the maximum nesting of brackets in their types. The measure of a goal cannot decrease for ever, as the number of assumptions is bounded by $(\#\mathrm{Sub}[S,T])^2$, and the second component must stay greater than zero. Thus the algorithm must terminate. ■

5.4 Completeness

In this section we need another technical construction. Call the subtype-algorithm statement $\Sigma \ \vdash_{\mathrm{a}} \ S \leq T$ *sound* if

- $\text{Tree}(S) \leq_{\text{tr}} \text{Tree}(T)$
- $\text{Tree}(U_1) \leq_{\text{tr}} \text{Tree}(U_2)$ for each $U_1 \leq U_2 \in \Sigma$.

Lemma 5.3 *If a statement, $\Sigma \models_{\text{a}} S \leq T$, is sound, then it matches the antecedent of one of the rules of the subtyping algorithm, and the premises of the rule are all sound.*

Proof: Let $\Sigma \models_{\text{a}} S \leq T$ be a sound statement. If $S = T$ then the reflexivity rule applies and there are no premises. If $S \leq T \in \Sigma$ then the assumption rule applies and there are no premises.

Suppose $S \leq T \notin \Sigma$. Say neither S nor T are of recursive form. Since the statement is sound, $\text{Tree}(S) \leq_{\text{tr}} \text{Tree}(T)$, so there is some tree simulation $\mathcal{R}$ such that $(\text{Tree}(S), \text{Tree}(T)) \in \mathcal{R}$. For this to be the case, $\mathcal{R}$ must satisfy the conditions of a tree simulation, and we consider those conditions as they apply to $\text{Tree}(T)(\Lambda)$.

If $T = \texttt{Long}$, then $\text{Tree}(T)(\Lambda) = \texttt{Long}^0$, and hence $\text{Tree}(S)(\Lambda) = \texttt{Long}^0$, so $S = \texttt{Long}$. Therefore, $S = T$ and we have dealt with this case above.

We pick $T = \texttt{InCh}\langle T_1, \ldots T_n\rangle$ as an example of the other non-recursive cases. They can all be tackled in a similar way. If $T = \texttt{InCh}\langle T_1, \ldots T_n\rangle$, then $\text{Tree}(T)(\Lambda) = \text{InCh}^n$. Hence, by the definition of Oompa tree simulation, either $\text{Tree}(S)(\Lambda) = \text{InCh}^n$ or $\text{Tree}(S)(\Lambda) = \text{Chan}^n$. It follows that S is either $\texttt{InCh}\langle S_1, \ldots S_n\rangle$ or $\texttt{Chan}\langle S_1, \ldots S_n\rangle$, and that $\text{Tree}(S_i) = \text{Tree}(S)(i) \leq_{\text{tr}} \text{Tree}(T)(i) = \text{Tree}(T_i)$. Either the InCh-InCh or the Chan-InCh rule rule will match $\Gamma \models_{\text{a}} S \leq T$, and the premises will all have the form $\Sigma \models_{\text{a}} S_i \leq T_i$. Σ is the same as for the antecedent, so the premises are sound as required.

Suppose one of S or T is of recursive form, say $S = \texttt{rec}\ s.S'$. This rule has one premise, of the form $\Sigma \cup \{S \leq T\} \models_{\text{a}} S'\{\!|\texttt{rec}\ s.S'/s|\!\} \leq T$. The new assumption, $S \leq T$, has $\text{Tree}(S) \leq_{\text{tr}} \text{Tree}(T)$ since the original judgement was sound. Also, $\text{Tree}(\texttt{rec}\ s.S') = \text{Tree}(S'\{\!|\texttt{rec}\ s.S'/s|\!\})$, and the right hand side of the judgements, T, hasn't changed, so the trees of the premise types are still the same. Thus the premise is sound. ■

Lemma 5.4 *Say S and T are closed types, and* $\text{Tree}(S) \leq_{\text{tr}} \text{Tree}(T)$. *Then* $\models_{\text{a}} S \leq T$.

Proof: If $\text{Tree}(T) \leq_{\text{tr}} \text{Tree}(S)$ then $\models_{\text{a}} S \leq T$ is sound. The algorithm cannot return false on a sound statement since, by Lemma 5.3, any premises of any sound statement are also sound, and hence matched by a rule. Since the algorithm is terminating by Theorem 5.2, it must return true. ■

Theorem 5.5 ***(Completeness)*** *If $\Gamma \models S \leq T$ then $\Gamma \vdash S \leq T$.*

Proof: $\Gamma \models S \leq T$ means $\mathrm{Tree}_\Gamma(S) \leq_{\mathrm{tr}} \mathrm{Tree}_\Gamma(T)$. But by Theorem 4.3,

$$\mathrm{Tree}(\mathrm{E}^\emptyset_\Gamma(S)) = \mathrm{Tree}_\Gamma(S) \leq_{\mathrm{tr}} \mathrm{Tree}_\Gamma(T) = \mathrm{Tree}(\mathrm{E}^\emptyset_\Gamma(T))$$

Both of $\mathrm{E}^\emptyset_\Gamma(S)$ and $\mathrm{E}^\emptyset_\Gamma(T)$ are closed, so by the Lemma 5.4, the algorithm will affirm $\vdash_{\mathrm{a}} \mathrm{E}^\emptyset_\Gamma(S) \leq \mathrm{E}^\emptyset_\Gamma(T)$ which is what $\Gamma \vdash S \leq T$ means. ■

5.5 Soundness

The following lemma describes a property usually called *weakening*.

Lemma 5.6 *$\Sigma \vdash_{\mathrm{a}} S \leq T$ implies $\Sigma \cup \Sigma' \vdash_{\mathrm{a}} S \leq T$.*

Proof: This is obvious after an inspection of the rules. ■

Lemma 5.7

1. *Suppose $\vdash_{\mathrm{a}} \mathtt{rec}\ s.S \leq T$ and $(\mathtt{rec}\ s.S \leq T) \notin \Sigma$. Then*

$$\Sigma \cup \{\mathtt{rec}\ s.S \leq T\} \vdash_{\mathrm{a}} U_1 \leq U_2 \quad \textit{implies} \quad \Sigma \vdash_{\mathrm{a}} U_1 \leq U_2$$

2. *Suppose $\vdash_{\mathrm{a}} S \leq \mathtt{rec}\ t.T$ and $(S \leq \mathtt{rec}\ t.T) \notin \Sigma$. Then*

$$\Sigma \cup \{S \leq \mathtt{rec}\ t.T\} \vdash_{\mathrm{a}} U_1 \leq U_2 \quad \textit{implies} \quad \Sigma \vdash_{\mathrm{a}} U_1 \leq U_2$$

Proof: We only prove this for the first case. Use induction on the length of a derivation of $\Sigma \cup \{\mathtt{rec}\ s.S \leq T\} \vdash_{\mathrm{a}} U_1 \leq U_2$.

Say the last rule used was Assumption. There are two cases. If $U_1 \leq U_2 \in \Sigma$, then the rule still applies with the smaller premise Σ. Say $U_1 = \mathtt{rec}\ s.S$ and $U_2 = T$. Now, we know $\vdash_{\mathrm{a}} \mathtt{rec}\ s.S \leq T$, so using the Lemma 5.6 we have $\Sigma \vdash_{\mathrm{a}} \mathtt{rec}\ s.S \leq T$.

In the cases where the last rule is not assumption, the premises are all of the form $\Sigma' \cup \{\mathtt{rec}\ s.S \leq T\} \vdash_{\mathrm{a}} U_1' \leq U_2'$. Any assumption added in Σ' not in Σ will definitely not be $\mathtt{rec}\ s.S \leq T$, therefore $\mathtt{rec}\ s.S \leq T \notin \Sigma'$. Thus the induction hypothesis applies to the premise, giving us $\Sigma' \vdash_{\mathrm{a}} U_1' \leq U_2'$. The last rule will still apply, giving us $\Sigma \vdash_{\mathrm{a}} U_1 \leq U_2$ as required. ■

Lemma 5.8

1. *If $\vdash_{\mathrm{a}} \mathtt{rec}\ s.S \leq T$ then $\vdash_{\mathrm{a}} S\{\!|^{\mathtt{rec}\ s.S}/_{s}|\!\} \leq T$*

2. *If $\vdash_{\mathrm{a}} S \leq \mathtt{rec}\ t.T$ and S isn't of recursive form, then $\vdash_{\mathrm{a}} S \leq T\{\!|^{\mathtt{rec}\ t.T}/_{t}|\!\}$*

3. If $\vdash_{\mathrm{a}} \mathtt{rec}\ s.S \leq \mathtt{rec}\ t.T$ *then* $\vdash_{\mathrm{a}} S\{\!|^{\mathtt{rec}\ s.S}/_{s}|\!\} \leq T\{\!|^{\mathtt{rec}\ t.T}/_{T}|\!\}$

Proof: 1. Consider the last rule used in $\vdash_{\mathrm{a}} \mathtt{rec}\ s.S \leq T$. It is either Reflexivity or Left Recursion and we consider the Reflexivity case first, so $T = \mathtt{rec}\ s.S$. Also by Reflexivity, we have

$$\{S\{\!|^{\mathtt{rec}\ s.S}/_{s}|\!\} \leq \mathtt{rec}\ s.T\} \vdash_{\mathrm{a}} S\{\!|^{\mathtt{rec}\ s.S}/_{s}|\!\} \leq S\{\!|^{\mathtt{rec}\ s.S}/_{s}|\!\}$$

Using the Right Recursion rule, this gives us

$$\vdash_{\mathrm{a}} S\{\!|^{\mathtt{rec}\ s.S}/_{s}|\!\} \leq \mathtt{rec}\ s.S$$

as required in this case.

Next we consider the case where the last rule used was Left Recursion. So, the tree had the form:

$$\frac{\{\mathtt{rec}\ s.S \leq T\} \vdash_{\mathrm{a}} S\{\!|^{\mathtt{rec}\ s.S}/_{s}|\!\} \leq T}{\vdash_{\mathrm{a}} \mathtt{rec}\ s.S \leq T}$$

We can use Lemma 5.7 to remove the assumption from the antecedent, giving us $\vdash_{\mathrm{a}} S\{\!|^{\mathtt{rec}\ s.S}/_{s}|\!\} \leq T$ as required in this case.

2. Similarly to part 1, the two cases which can apply are Reflexivity and Right Recursion, and they can be dealt with in the same way. The extra condition on this case, means we can avoid considering the case where the last rule was Left Recursion.

3. We use part 1 and then part 2 to give the result.

■

Lemma 5.9 *If* S *and* T *are closed and guarded types such that* $\vdash_{\mathrm{a}} S \leq T$*, then* $\mathrm{Tree}(S) \leq_{\mathrm{tr}} \mathrm{Tree}(T)$*.*

Proof: Let

$$\mathcal{R} = \{(\mathrm{Tree}(S), \mathrm{Tree}(T)) | \ \vdash_{\mathrm{a}} S \leq T \text{ where } S \text{ and } T \text{ are guarded}\}$$

We claim that $\mathcal{R}$ is a tree simulation.

Assume $(\mathrm{Tree}(S), \mathrm{Tree}(T)) \in \mathcal{R}$. We go through the conditions of what it means to be a tree simulation, depending on the construction of T. Say that $T = \mathtt{Chan}\langle T_1, \ldots T_n\rangle$. Then $\mathrm{Tree}(T)(\Lambda) = \mathrm{Chan}^n$. The only rules that could establish $\vdash_{\mathrm{a}} S \leq T$ is either Reflexivity or the Channel-Channel subtyping

rule. The first case is obvious, the second means that $S = \texttt{Chan}\langle S_1, \ldots S_n\rangle$. Thus $\text{Tree}(S)(\Lambda) = \text{Chan}^n$.

For $\mathcal{R}$ to work, we need both

$$(\text{Tree}(S)(i), \text{Tree}(T)(i)) \in \mathcal{R}$$
$$(\text{Tree}(T)(i), \text{Tree}(S)(i)) \in \mathcal{R}$$

Now $\text{Tree}(S)(i) = \text{Tree}(S_i)$ and $\text{Tree}(T)(i) = \text{Tree}(T_i)$. The premises of the rule give us $\vdash_{\text{a}} S_i \leq T_i$ and $\vdash_{\text{a}} T_i \leq S_i$. By Lemma 5.8 we can unfold S_i and T_i into guarded terms S_i' and T_i' such that $\vdash_{\text{a}} S_i' \leq T_i'$. $\vdash_{\text{a}} S_i' \leq T_i'$ and $\vdash_{\text{a}} T_i' \leq S_i'$, so $(\text{Tree}(S_i'), \text{Tree}(T_i')) \in \mathcal{R}$ and $(\text{Tree}(T_i'), \text{Tree}(S_i')) \in \mathcal{R}$. By the definition of the Tree function, unfolding a type doesn't affect it's tree, so we have $(\text{Tree}(S)(i), \text{Tree}(T)(i)) \in \mathcal{R}$ and $(\text{Tree}(T)(i), \text{Tree}(S)(i)) \in \mathcal{R}$. This gives us this case.

Since S and T are guarded the other cases are all very similar to the one given. ■

Theorem 5.10 ***(Soundness)*** *If* $\Gamma \vdash S \leq T$ *then* $\Gamma \models S \leq T$.

Proof: $\Gamma \vdash S \leq T$ means $\vdash_{\text{a}} \text{E}_\Gamma^\emptyset(S) \leq \text{E}_\Gamma^\emptyset(T)$. $\text{E}_\Gamma^\emptyset(S)$ and $\text{E}_\Gamma^\emptyset(T)$ are closed and by Lemma 5.8 we can unfold them to give guarded types S' and T' such that $\vdash_{\text{a}} S' \leq T'$. By Lemma 5.9, this gives us $\text{Tree}(S') \leq_{\text{tr}} \text{Tree}(T')$.

Now, by the definition of the Tree function, unfolding doesn't affect trees, so Theorem 4.3 gives us $\text{Tree}_\Gamma(S) \leq_{\text{tr}} \text{Tree}_\Gamma(T)$, which is what $\Gamma \models S \leq T$ means. ■

6 The Validity of the Type Safety System

We need to show that if a definition set passes our type safety test, then the systems which develop from its initial system never commit a type violation. This is usually called the *soundness* of a type system. In order to talk meaningfully about a system behaving according to the type system, it is useful to have a concept of the state of the type system "at run time". We augment the type safety system with rules that allow us to state that an Oompa system is obeying the type discipline. We call this the *dynamic typing system.*

We use this concept to validate our type safety system. We show that a type safe definition set gives rise to a dynamically type safe initial system. We also show that dynamic type safety is preserved by the operational semantics. The main result follows from observing that dynamically type safe systems are not committing a type violation.

This technique is called *subject reduction* [WF91], which considers "each intermediate state of a program is itself a program [...]. Thus, proving type soundness reduces to proving that well-typed programs yield only well-typed results." There are two details to consider when applying this approach to Oompa. First, our programs are written in static definitions, which are type checked, and our Oompa systems are not of the same form (this is why we will need a separate *dynamic* type safety system). Second, it is not useful to consider a process calculus reducing to a result, so we merely require all the systems arrived at by the operational semantics to be dynamically type safe.

We introduce an abbreviation to help with the readability of the proofs of the following theorems. Every object o given a type in the type dictionary must be given a class type. Say $o\colon C \in \Phi$, and say C's definition in Γ is

$$\texttt{class}\ C\ \{a_1\colon \mathrm{AtT}_1, \ldots a_n\colon \mathrm{AtT}_n, \mathrm{mdef}^*\}$$

Then we let Att_o be the set of attribute typings of o's class, i.e.

$$Att_o = \{a_1\colon \mathrm{AtT}_1, \ldots a_n\colon \mathrm{AtT}_n\}$$

6.1 Dynamic Type Safety

Given an Oompa system $\Gamma \rhd g\{^{\Phi}_{\Delta}$, we use the notation $\Gamma \models_{\mathrm{d}} g\{^{\Phi}_{\Delta}$ to mean that the system is type safe in a dynamic sense, i.e. all values in the system are being used appropriately. To speak about the type safety of agent code (as opposed to static method code) we need to modify the original type safety system slightly to take account of variable naming issues. The rules that have changed are New Channel, Create, Receive and Attribute Access:

$$\frac{\Gamma, \Phi \cup \{c'\colon \mathrm{ChT}\} \models_{\mathrm{d}} p\{|^{c'}/c|\}}{\Gamma, \Phi \models_{\mathrm{d}} \texttt{new}\ c\colon \mathrm{ChT}\ p} \qquad \frac{\Gamma, \Phi \cup \{o'\colon C\} \models_{\mathrm{d}} p\{|^{o'}/o|\}}{\Gamma, \Phi \models_{\mathrm{d}} \texttt{create}\ o\colon C\ p}$$

$$\frac{\Gamma, \Phi \vdash c\colon \texttt{InCh}\langle T_1, \ldots T_n\rangle \quad \Gamma, \Phi \cup \{r'_1\colon T_1, \ldots r'_n\colon T_n\} \models_{\mathrm{d}} p\{|^{r'_1}/r_1 \ldots {}^{r'_n}/r_n|\}}{\Gamma, \Phi \models_{\mathrm{d}} c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p}$$

$$\frac{\Gamma, \Phi \vdash a\colon \texttt{Attr}\{T\} \qquad \Gamma, \Phi \cup \{r'\colon T\} \models_{\mathrm{d}} p\{|^{r'}/r|\}}{\Gamma, \Phi \models_{\mathrm{d}} a?r\ p}$$

where in the Create rule C is defined as a class in Γ.

The dynamic type safety of an Oompa system is defined in terms of its parts, as follows:

- We say $\Gamma \models_{\mathrm{d}} g\{^{\Phi}_{\Delta}$ if and only if $\Gamma, \Phi \models_{\mathrm{d}} g$ and $\Gamma, \Phi \models_{\mathrm{d}} \Delta$

- We say $\Gamma, \Phi \vdash_{\mathrm{d}} g$ if and only if $g = \texttt{nil}$ or $g = o_1[p_1] \mid \ldots o_n[p_n]$ and $\Gamma, \Phi \vdash_{\mathrm{d}} o_i[p_i]$ for all i.
- We say $\Gamma, \Phi \vdash_{\mathrm{d}} o[p]$ if and only if $\Phi(o) = C$ for some class C such that

$$\texttt{class}\ \ C\ \{a_1{:}\,\mathrm{AtT}_1\ \cdots\ a_n{:}\,\mathrm{AtT}_n\ \ mdef^*\} \in \Gamma$$

  and $\Gamma, \Phi \cup \{a_1{:}\,\mathrm{AtT}_1 \ldots a_n{:}\,\mathrm{AtT}_n\} \vdash_{\mathrm{d}} p$
- We say $\Gamma, \Phi \vdash_{\mathrm{d}} \Delta$ if and only if for all $o \in \mathrm{Dom}(\Delta)$ we have $\Phi(o) = C$ for some class C such that

$$\texttt{class}\ \ C\ \{a_1{:}\,\texttt{Attr}\{T_1\}\ \cdots\ a_n{:}\,\texttt{Attr}\{T_n\}\ \ mdef^*\} \in \Gamma$$

  and for all $a \in \mathrm{Dom}(\Delta(o))$ we have $a = a_i$ for some i and $\Gamma, \Phi\ \vdash o(a){:}\,T_i$

The key property of dynamic type safety is that a Oompa system which is dynamically type safe is clearly not type violating.

Lemma 6.1 *If* $\Gamma \vdash_{\mathrm{d}} g\{^{\Phi}_{\Delta}$ *then* $\Gamma\ \rhd\ g\{^{\Phi}_{\Delta}$ *isn't type violating.*

Proof: Say $o[p]$ is an agent in g. We know $\Gamma \vdash_{\mathrm{d}} g\{^{\Phi}_{\Delta}$, so $\Gamma, \Phi \vdash_{\mathrm{d}} g$ and therefore $\Gamma, \Phi \vdash_{\mathrm{d}} o[p]$. We consider the form of $o[p]$. Say that $o[p]$ is invoking a method m of an object o', i.e. $p = o'.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_{n'})\ p'$. Thus we have

$$\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} o'.m!\langle v_1, \ldots v_n\rangle?(r_1, \ldots r_{n'})\ p'$$

By the rules of the dynamic type safety system, this means that we must have $\Gamma, \Phi \cup Att_o\ \vdash\ o'{:}\,\texttt{Intf}\{m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle\}$ and for each $0 \leq i \leq n$, $\Gamma, \Phi \cup Att_o\ \vdash\ v_i{:}\,T_i$, since they are premises for the rule for checking invocations.

Now from the first statement, we can conclude that o''s class in Φ must have a method m with signature type $m?(S_1, \ldots S_n)!\langle S'_1 \ldots S'_{n'}\rangle$, and by the rules of the typing system, we will have that $\Gamma, \Phi\ \vdash\ T_i \leq S_i$. Thus the agent cannot be performing a feature or arity mismatch. Using the Subtyping rule of the typing system, we have

$$\frac{\Gamma\ \vdash\ T_i \leq S_i \qquad \Gamma, \Phi \cup Att_o\ \vdash\ v_i{:}\,T_i}{\Gamma, \Phi \cup Att_o\ \vdash\ v_i{:}\,S_i}$$

Hence the agent isn't performing a value mismatch. This means that the agent isn't type violating.

The cases for Send and Receive are similar. Next, we consider Attribute Update. Say $o[p]$ is updating an attribute a, i.e. $p = a!v\ p'$. Thus we have

$$\Gamma, \Phi \cup Att_o \models_{\mathrm{d}} a!v\ p'$$

By the rules of the dynamic type safety system, we must have $\Gamma, \Phi \cup Att_o \vdash a{:}\,\texttt{Attr}\{T\}$ and $\Gamma, \Phi \cup Att_o \vdash v{:}\,T$. Thus, we know that o's class must have an attribute a of type $\texttt{Attr}\{T\}$. Thus the agent isn't performing either a feature mismatch, or a value mismatch.

The case for Attribute Access is similar. It follows that no agent in the system is type violating, so $\Gamma \rhd g\{^{\Phi}_{\Delta}$ isn't type violating. ■

Lemma 6.2 *If* $\Gamma, \Phi \vdash p$ *then* $\Gamma, \Phi \models_{\mathrm{d}} p$.

Proof: A proof tree in the static type safety system is also a proof tree in the dynamic type safety system. In each of the four rules which have changed, we can use an identity substitution to justify instances of their standard form in the dynamic type safety system. ■

Another important property is that the initial system of a statically type safe definition set is dynamically type safe.

Lemma 6.3 *If* $\Phi_0 \vdash \Gamma$ *then* $\Gamma \models_{\mathrm{d}} g_\Gamma\{^{\Phi_0}_{\emptyset}$.

Proof: We know that the state is fine, since it is empty. We have to consider the initial agents. Now $g = \mathrm{dummy}_1[p_1] \mid \ldots \mathrm{dummy}_n[p_n]$ and C_i has no attributes, by the requirements for the initial system. Also, p_i doesn't use `return`. Since $\Phi_0 \vdash \Gamma$, we know $\Gamma, \Phi_0 \vdash C_i$ for all i. Thus, $\Gamma, \Phi_0 \vdash \mathrm{main\_def}$ where main_def is the method definition of the main method of C_i. Hence $\Gamma, \Phi_0 \vdash p_i$, and by Lemma 6.2 we have $\Gamma, \Phi_0 \models_{\mathrm{d}} p_i$. Since the class C_i has no attributes $Att_{\mathrm{dummy}_i} = \emptyset$ giving us $\Gamma, \Phi_0 \models_{\mathrm{d}} \mathrm{dummy}_i[p_i]$. Thus $\Gamma, \Phi_0 \models_{\mathrm{d}} g_\Gamma$, so $\Gamma \models_{\mathrm{d}} g_\Gamma\{^{\Phi_0}_{\emptyset}$. ■

6.2 Preservation of Dynamic Type Safety

We know that a dynamically type safe system is not type violating, and we know that a statically type safe definition set starts out dynamically type safe. To guarantee that a statically type safe definition set won't become type violating, it is sufficient to show that the operational semantics preserve dynamic type safety.

First, however, we need to prove certain properties of the dynamic type safety system.

Lemma 6.4

1. Weaken: *If we have* $\Gamma, \Phi \vdash_{\mathrm{d}} p$ *and* $\Phi \cap \Phi' = \emptyset$*, then* $\Gamma, \Phi \cup \Phi' \vdash_{\mathrm{d}} p$.
2. Strengthen: *If we have* $\Gamma, \Phi \cup \Phi' \vdash_{\mathrm{d}} p$ *and none of the variables in* Φ' *occur in* p*, then* $\Gamma, \Phi \vdash_{\mathrm{d}} p$.
3. Switch: *If* $\Gamma, \Phi \cup \{v{:}T\} \vdash_{\mathrm{d}} p$ *and* $\Gamma, \Phi \vdash v'{:}T$ *, then* $\Gamma, \Phi \cup \{v{:}T\} \vdash_{\mathrm{d}} p\{^{v'}/_{v}\}$.
4. Rename: *If* $\Gamma, \Phi \cup \{v{:}T\} \vdash_{\mathrm{d}} p$ *then* $\Gamma, \Phi \cup \{v'{:}T\} \vdash_{\mathrm{d}} p\{^{v'}/_{v}\}$ *for new* v'.
5. Weaken 2: *If* $\Gamma, \Phi \cup \{v{:}T\} \vdash_{\mathrm{d}} p$ *and* $\Gamma \vdash T' \leq T$ *then* $\Gamma, \Phi \cup \{v{:}T'\} \vdash_{\mathrm{d}} p$.

Proof: Cases 1–3 can be proved by a standard inductive argument on the structure of proof trees.

Case 4 is proved in terms of the first three. Say $\Gamma, \Phi \cup \{v{:}T\} \vdash_{\mathrm{d}} p$ then $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T\} \vdash_{\mathrm{d}} p$ by Weaken. $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T\} \vdash v'{:}T$ by Assumption, so $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T\} \vdash_{\mathrm{d}} p\{^{v'}/_{v}\}$ by Switch and $\Gamma, \Phi \cup \{v'{:}T\} \vdash_{\mathrm{d}} p\{^{v'}/_{v}\}$ by Strengthen.

Case 5 is proved in terms of the first four. Say $\Gamma, \Phi \cup \{v{:}T\} \vdash_{\mathrm{d}} p$ and $\Gamma \vdash T' \leq T$. Then $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T'\} \vdash_{\mathrm{d}} p$ for some new v' by Weaken. By Assumption, we have $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T'\} \vdash v'{:}T'$ and by Subtyping, we have $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T'\} \vdash v'{:}T$. We use Switch to give us $\Gamma, \Phi \cup \{v{:}T\} \cup \{v'{:}T'\} \vdash_{\mathrm{d}} p\{^{v'}/_{v}\}$. Next, Strengthen to $\Gamma, \Phi \cup \{v'{:}T'\} \vdash_{\mathrm{d}} p\{^{v'}/_{v}\}$ and Rename v' back to v to give $\Gamma, \Phi \cup \{v{:}T'\} \vdash_{\mathrm{d}} p$. ■

Lemma 6.5 *If we have* $\Gamma, \Phi \vdash_{\mathrm{d}} \Delta$ *and* $\Phi \cap \Phi' = \emptyset$*, then* $\Gamma, \Phi \cup \Phi' \vdash_{\mathrm{d}} \Delta$.

Proof: Trivial.

Theorem 6.6 *If* $\Phi_0 \vdash \Gamma$ *and* $\Gamma \vdash_{\mathrm{d}} g\{^{\Phi}_{\Delta}$ *and* $\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}$ *then* $\Gamma \vdash_{\mathrm{d}} g'\{^{\Phi'}_{\Delta'}$

Proof: We use induction on the derivation of $\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}$. We consider the last rule used in this derivation. First, we deal with the inductive cases.

Left Equivalence

Consider the case where the last rule used was Left Equivalence. Then the tree is structured as follows:

$$\dfrac{g'' \equiv g \qquad \dfrac{\vdots}{\Gamma \rhd g''\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}}}{\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}}$$

The equivalence of g and g'' means that they differ only in the reordering and regrouping of their constituent agents. This reordering and regrouping will not affect type safety, so we have $\Gamma \vdash_{\mathrm{d}} g''\{^{\Phi}_{\Delta}$. Thus, the inductive hypothesis applies to the right-hand subtree, giving us $\Gamma \vdash_{\mathrm{d}} g'\{^{\Phi'}_{\Delta'}$, as required.

Right Equivalence

If the last rule used was Right Equivalence, then the tree is structured like:

$$\dfrac{g'' \equiv g' \qquad \dfrac{\vdots}{\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g''\{^{\Phi'}_{\Delta'}}}{\Gamma \rhd g\{^{\Phi}_{\Delta} \longrightarrow g'\{^{\Phi'}_{\Delta'}}$$

The induction hypothesis will apply to the right-hand subtree, giving us $\Gamma \vdash_{\mathrm{d}} g''\{^{\Phi'}_{\Delta'}$. However, by the equivalence of g'' and g', we have $\Gamma \vdash_{\mathrm{d}} g'\{^{\Phi'}_{\Delta}$.

Parallelism

If the last rule used was Parallelism then the tree has the structure:

$$\dfrac{\dfrac{\vdots}{\Gamma \rhd g_1\{^{\Phi}_{\Delta} \longrightarrow g_1'\{^{\Phi'}_{\Delta'}}}{\Gamma \rhd (g_1 \mid g_2)\{^{\Phi}_{\Delta} \longrightarrow (g_1' \mid g_2)\{^{\Phi'}_{\Delta'}}$$

where $g = (g_1 \mid g_2)$ and $g' = (g_1' \mid g_2)$. By the premise of the theorem, we have $\Gamma \vdash_{\mathrm{d}} (g_1 \mid g_2)\{^{\Phi}_{\Delta}$ giving us $\Gamma, \Phi \vdash_{\mathrm{d}} (g_1 \mid g_2)$ and $\Gamma, \Phi \vdash_{\mathrm{d}} \Delta$. So we have $\Gamma, \Phi \vdash_{\mathrm{d}} g_1$ and $\Gamma, \Phi \vdash_{\mathrm{d}} g_2$. Thus the inductive hypothesis applies to the subtree, giving us $\Gamma \vdash_{\mathrm{d}} g_1'\{^{\Phi'}_{\Delta'}$. Therefore $\Gamma, \Phi' \vdash_{\mathrm{d}} g_1'$ and $\Gamma, \Phi' \vdash_{\mathrm{d}} \Delta'$. The type dictionary, Φ, is only enlarged by the operational semantics, so we can use Lemma 6.4, case 1, to weaken $\Gamma, \Phi \vdash_{\mathrm{d}} g_2$ to give us $\Gamma, \Phi' \vdash_{\mathrm{d}} g_2$. Hence $\Gamma, \Phi' \vdash_{\mathrm{d}} (g_1' \mid g_2)$. Thus we have $\Gamma \vdash_{\mathrm{d}} g'\{^{\Phi'}_{\Delta'}$ as required.

End

Next we consider the axioms of which End is the first. This case is trivial as `nil` is a type safe agent, and the state remains unchanged.

Fork

Next we consider Fork, which has the structure

$$\Gamma \ \rhd \ o[\texttt{fork}\{p_1\}\ p_2]\{^{\Phi}_{\Delta} \ \longrightarrow \ o[p_1] \mid o[p_2]\{^{\Phi}_{\Delta}$$

By the premise of the theorem $\Gamma \vdash_{\mathrm{d}} o[\texttt{fork}\{p_1\}p_2]\{^{\Phi}_{\Delta}$ so $\Gamma, \Phi \vdash_{\mathrm{d}} o[\texttt{fork}\{p_1\}p_2]$ and $\Gamma, \Phi \vdash_{\mathrm{d}} \Delta$. From the former we have $\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} \texttt{fork}\{p_1\}p_2$. In order for this to be a theorem in the dynamic type safety system, the proof tree must have the following structure:

$$\frac{\dfrac{\vdots}{\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} p_1} \qquad \dfrac{\vdots}{\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} p_2}}{\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} \texttt{fork}\{p_1\}p_2}$$

Therefore $\Gamma, \Phi \vdash_{\mathrm{d}} o[p_1]$ and $\Gamma, \Phi \vdash_{\mathrm{d}} o[p_2]$ and thus $\Gamma, \Phi \vdash_{\mathrm{d}} (o[p_1] \mid o[p_2])$ The state remains unchanged, so $\Gamma \vdash_{\mathrm{d}} (o[p_1] \mid o[p_2])\{^{\Phi}_{\Delta}$, as required.

New Channel

The next axiom to consider is New Channel. This axiom has the form

$$\Gamma \ \rhd \ o[\texttt{new}\ c\colon \mathrm{ChT}\ p]\{^{\Phi}_{\Delta} \ \longrightarrow \ o[p\{|^{c'}/_{c}|\}]\{^{\Phi \cup \{c'\colon \mathrm{ChT}\}}_{\Delta}$$

where c' is the new channel name "chosen" by the axiom. By the premise of the theorem, we have $\Gamma \vdash_{\mathrm{d}} o[\texttt{new}\ c\colon \mathrm{ChT}\ p]\{^{\Phi}_{\Delta}$, so $\Gamma, \Phi \vdash_{\mathrm{d}} o[\texttt{new}\ c\colon \mathrm{ChT}\ p]$, and $\Gamma, \Phi \vdash_{\mathrm{d}} \Delta$. Thus we have $\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} \texttt{new}\ c\colon \mathrm{ChT}\ p$. The dynamic type safety system gives us $\Gamma, \Phi \cup Att_o \cup \{c''\colon \mathrm{ChT}\} \vdash_{\mathrm{d}} p\{|^{c''}/_{c}|\}$. By Lemma 6.4, case 4, we can use a different new name, so we pick c', giving us $\Gamma, \Phi \cup Att_o \cup \{c'\colon \mathrm{ChT}\} \vdash_{\mathrm{d}} p\{|^{c'}/_{c}|\}$. Then $\Gamma, \Phi \cup \{c'\colon \mathrm{ChT}\} \vdash_{\mathrm{d}} o[p\{|^{c'}/_{c}|\}]$. We can weaken $\Gamma, \Phi \vdash_{\mathrm{d}} \Delta$ to give $\Gamma, \Phi \cup \{c'\colon \mathrm{ChT}\} \vdash_{\mathrm{d}} \Delta$ by Lemma 6.5. Hence $\Gamma \vdash_{\mathrm{d}} o[p\{|^{c'}/_{c}|\}]\{^{\Phi \cup \{c'\colon \mathrm{ChT}\}}_{\Delta}$ as required.

Communication

The next axiom we consider is Communication. This axiom has the structure:

$$\begin{array}{rcll} \Gamma & \rhd & o_1[c!\langle v_1, \ldots v_n\rangle\ p_1] \ \mid & \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right. \\ & & o_2[c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_2] & \\ & & & \\ & \longrightarrow & o_1[p_1] \ \mid & \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right. \\ & & o_2[p_2\{|^{v_1}/_{r_1}, \ldots {}^{v_n}/_{r_n}|\}] & \end{array}$$

By the premise of the theorem, we have

$$\Gamma \vdash_{\mathrm{d}} o_1[c!\langle v_1, \ldots v_n\rangle\ p_1] \mid o_2[c?(r_1{:}\,T_1, \ldots r_n{:}\,T_n)\ p_2]\{^{\Phi}_{\Delta}$$

so we must have both

$$\Gamma, \Phi \cup Att_{o_1} \vdash_{\mathrm{d}} c!\langle v_1, \ldots v_n\rangle\ p_1$$
$$\Gamma, \Phi \cup Att_{o_2} \vdash_{\mathrm{d}} c?(r_1{:}\,T_1, \ldots r_n{:}\,T_n)\ p_2$$

The first of these, by the laws of dynamic type safety, gives us $\Gamma, \Phi \cup Att_{o_1} \vdash c{:}\,\mathtt{OuCh}\langle S_1 \ldots S_n\rangle$ and $\Gamma, \Phi \cup Att_{o_1} \vdash v_i{:}\,S_i$, and the second gives us $\Gamma, \Phi \cup Att_{o_2} \vdash c{:}\,\mathtt{InCh}\langle T_1 \ldots T_n\rangle$. The only way we could assign these two types to c in the typing system would be if $\Gamma \vdash S_i \leq T_i$ so we also have $\Gamma, \Phi \cup Att_{o_1} \vdash v_i{:}\,T_i$ using the Subtyping rule of the typing system.

The proof tree of $\Gamma, \Phi \cup Att_{o_2} \vdash_{\mathrm{d}} c?(r_1{:}\,T_1, \ldots r_n{:}\,T_n)\ p_2$ has a right subtree which concludes $\Gamma, \Phi \cup Att_{o_2} \cup \{r'_1{:}\,T_1 \ldots r'_n{:}\,T_n\} \vdash_{\mathrm{d}} p_2\{|^{r'_1}/_{r_1} \ldots {}^{r'_n}/_{r_n}|\}$. Then, since the v_i have suitable typings in Φ, we can apply Lemma 6.4, case 3 and case 2 to use the v_i instead of the r_i to give $\Gamma, \Phi \cup Att_{o_2} \vdash_{\mathrm{d}} p_2\{|^{v_1}/_{r_1} \ldots {}^{v_n}/_{r_n}|\}$. Thus $\Gamma, \Phi \vdash_{\mathrm{d}} o_2[p_2\{|^{v_1}/_{r_1} \ldots {}^{v_n}/_{r_n}|\}]$.

Also, $\Gamma, \Phi \cup Att_{o_1} \vdash_{\mathrm{d}} c!\langle v_1, \ldots v_n\rangle\ p_1$ has a right subtree which concludes $\Gamma, \Phi \cup Att_{o_1} \vdash_{\mathrm{d}} p_1$ so $\Gamma, \Phi \vdash_{\mathrm{d}} o_1[p_1]$. Using this and the above result, gives us $\Gamma, \Phi \vdash_{\mathrm{d}} o_1[p_1] \mid o_2[p_2\{|^{v_1}/_{r_1} \ldots {}^{v_n}/_{r_n}|\}]$. The state remains unchanged, so we have $\Gamma \vdash_{\mathrm{d}} o_1[p_1] \mid o_2[p_2\{|^{v_1}/_{r_1} \ldots {}^{v_n}/_{r_n}|\}]\{^{\Phi}_{\Delta}$ as required.

Method Invocation

The next axiom is Method Invocation. This axiom has the form:

$$\Gamma \quad \rhd \quad o[o_1.m!\langle v_1, \ldots v_l\rangle?(s_1, \ldots s_n)\ p] \qquad \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right.$$

$$\longrightarrow \quad \begin{array}{l} o[r?(s_1{:}\,T_1, \ldots s_n{:}\,T_n)\ p] \ \mid \\ o_1[p_1\{|^{v_1}/_{r_1}, \ldots {}^{v_l}/_{r_l}, {}^{o_1}/_{\mathtt{this}}, {}^{r}/_{\mathtt{return}}|\}] \end{array} \qquad \left\{ \begin{array}{l} \Phi' \\ \Delta \end{array} \right.$$

where $\Phi' = \Phi \cup \{r{:}\,\mathtt{Chan}\langle T_1, \ldots T_n\rangle\}$, and r is the return channel "chosen" by the axiom. The main premise gives us $\Gamma \vdash_{\mathrm{d}} o[o_1.m!\langle v_1, \ldots v_l\rangle?(s_1, \ldots s_n)\ p]\{^{\Phi}_{\Delta}$, so $\Gamma, \Phi \vdash_{\mathrm{d}} o[o_1.m!\langle v_1, \ldots v_l\rangle?(s_1, \ldots s_n)\ p]$, and hence we have

$$\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} o_1.m!\langle v_1, \ldots v_l\rangle?(s_1, \ldots s_n)\ p$$

From this, we can conclude the following three facts:

$$\Gamma, \Phi \cup Att_o \vdash o_1{:}\,\mathtt{Intf}\{m?(S_1, \ldots S_n)!\langle S'_1, \ldots S'_l\rangle\} \qquad \text{(A)}$$
$$\Gamma, \Phi \cup Att_o \cup \{r'{:}\,\mathtt{Chan}\langle S'_1 \ldots S'_n\rangle\} \vdash_{\mathrm{d}} r'?(s_1{:}\,S'_1, \ldots s_n{:}\,S'_n)\ p \qquad \text{(B)}$$
$$\Gamma, \Phi \cup Att_o \vdash v_i{:}\,S_i \qquad \text{(C)}$$

Now $o_1 : C \in \Phi$ and the fact that the Method Invocation axiom applied means that C must have a method with a signature of the form

$$m?(r_1 : U'_1 \dots r_l : U'_l)!\langle y_1 : T_1 \dots y_n : T_n \rangle$$

By (A), using the subtyping system, we know that

$$\Gamma \ \vdash \ m?(U'_1 \dots U'_l)!\langle T_1 \dots T_n \rangle \leq m?(S_1, \dots S_n)!\langle S'_1, \dots S'_l \rangle$$

which means that $\Gamma \ \vdash \ S_i \leq U'_i$ and $\Gamma \ \vdash \ T_i \leq S'_i$.

By (B), using the dynamic type safety rule for receive, we have

$$\Gamma, \Phi \cup Att_o \cup \{r' : \texttt{Chan}\langle S'_1 \dots S'_n \rangle\} \cup \{s'_1 : S'_1 \dots s'_n : S'_n\} \models_{\mathrm{d}} p\{|{}^{s'_1}/_{s_1} \dots {}^{s'_n}/_{s_n}|\}$$

By using Lemma 6.4, case 2, we can remove r''s typing from the context, to give $\Gamma, \Phi \cup Att_o \cup \{s'_1 : S'_1 \dots s'_n : S'_n\} \models_{\mathrm{d}} p\{|{}^{s'_1}/_{s_1} \dots {}^{s'_n}/_{s_n}|\}$. We can now use Lemma 6.4, case 5, to strengthen the typings of the s'_i, giving us $\Gamma, \Phi \cup Att_o \cup \{s'_1 : T_1 \dots s'_n : T_n\} \models_{\mathrm{d}} p\{|{}^{s'_1}/_{s_1} \dots {}^{s'_n}/_{s_n}|\}$. Now use Lemma 6.4, case 1, to give us $\Gamma, \Phi \cup Att_o \cup \{r : \texttt{Chan}\langle T_1 \dots T_n \rangle\} \cup \{s'_1 : T_1 \dots s'_n : T_n\} \models_{\mathrm{d}} p\{|{}^{s'_1}/_{s_1} \dots {}^{s'_n}/_{s_n}|\}$ from which, using the rules of the dynamic typing system, we can conclude $\Gamma, \Phi \cup Att_o \cup \{r : \texttt{Chan}\langle T_1 \dots T_n \rangle\} \models_{\mathrm{d}} r?(s_1 : T_1 \dots s_n : T_n)\ p$. Thus $\Gamma, \Phi \cup \{r : \texttt{Chan}\langle T_1 \dots T_n \rangle\} \models_{\mathrm{d}} o[r?(s_1 : T_1 \dots s_n : T_n)\ p]$.

By (C) we know that $\{v_i : U_i\} \in \Phi$ and that $\Gamma \ \vdash \ U_i \leq S_i$, so by transitivity $\Gamma \ \vdash \ U_i \leq U'_i$. Now we know that o_1 has a method with a signature $m?(r_1 : U'_1 \dots r_l : U'_l)!\langle y_1 : T_1 \dots y_n : T_n \rangle$ and if it has code p_1, we know that

$$\Gamma, \Phi_0 \cup \{r_1 : U'_1 \dots r_l : U'_l\} \cup \{\texttt{return} : \texttt{OuCh}\langle T_1 \dots T_n \rangle\} \cup \{\texttt{this} : C\} \ \vdash p_1$$

We bring this into the dynamic type safety system and apply Lemma 6.4 to give

$$\Gamma, \Phi_0 \cup \{v_1 : U_1 \dots v_l : U_l\} \cup \{r : \texttt{Chan}\langle T_1 \dots T_n \rangle\} \cup \{o_1 : C\} \models_{\mathrm{d}} p_1\{|{}^{v_1}/_{r_1} \dots {}^{v_l}/_{r_l}, {}^{r}/_{\texttt{return}}, {}^{o_1}/_{\texttt{this}}|\}$$

and again

$$\Gamma, \Phi \cup Att_o \cup \{r : \texttt{Chan}\langle T_1 \dots T_n \rangle\} \models_{\mathrm{d}} p_1\{|{}^{v_1}/_{r_1} \dots {}^{v_l}/_{r_l}, {}^{r}/_{\texttt{return}}, {}^{o_1}/_{\texttt{this}}|\}$$

Thus

$$\Gamma, \Phi \cup \{r : \texttt{Chan}\langle T_1 \dots T_n \rangle\} \models_{\mathrm{d}} o_1[p_1\{|{}^{v_1}/_{r_1} \dots {}^{v_l}/_{r_l}, {}^{r}/_{\texttt{return}}, {}^{o_1}/_{\texttt{this}}|\}]$$

Putting and the earlier fact together, we have

$$\Gamma, \Phi \cup \{r : \texttt{Chan}\langle T_1 \dots T_n \rangle\} \models_{\mathrm{d}} \begin{array}{l} o[r?(s_1 : T_1 \dots s_n : T_n)\ p] \mid \\ o_1[p_1\{|{}^{v_1}/_{r_1} \dots {}^{v_l}/_{r_l}, {}^{r}/_{\texttt{return}}, {}^{o_1}/_{\texttt{this}}|\}] \end{array}$$

The state hasn't changed, so we have $\Gamma \models_{\mathrm{d}} g'\{^{\Phi'}_{\Delta'}$ as required.

Object Creation

Next we have Object Creation, which has the following structure:

$$\Gamma \quad \rhd \quad o[\texttt{create}\ o_1{:}C\ p] \quad \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right.$$

$$\longrightarrow \quad o[p\{\!|{}^{o'_1}/_{o_1}|\!\}] \qquad \left\{ \begin{array}{l} \Phi \cup \{o'_1{:}C\} \\ \Delta \cup \{o'_1 = \emptyset\} \end{array} \right.$$

where o'_1 is the new object name "chosen" by the axiom. From the main premise of the theorem we have $\Gamma \vdash_{\mathrm{d}} o[\texttt{create}\ o_1{:}C\ p]\{^{\Phi}_{\Delta}$ which gives us both $\Gamma, \Phi \vdash_{\mathrm{d}} o[\texttt{create}\ o_1{:}C\ p]$ and $\Gamma, \Phi \vdash_{\mathrm{d}} \Delta$. So $\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} \texttt{create}\ o_1{:}C\ p$. This implies that $\Gamma, \Phi \cup Att_o \cup \{o''_1{:}C\} \vdash_{\mathrm{d}} p\{\!|{}^{o''_1}/_{o_1}|\!\}$. So by Lemma 6.4, case 4, we can use o'_1 instead o''_1 to give $\Gamma, \Phi \cup Att_o \cup \{o'_1{:}C\} \vdash_{\mathrm{d}} p\{\!|{}^{o'_1}/_{o_1}|\!\}$ so $\Gamma, \Phi \cup \{o'_1{:}C\} \vdash_{\mathrm{d}} o[p\{\!|{}^{o'_1}/_{o_1}|\!\}]$. The addition of the empty assignment to the typing state, will not affect it's dynamic type safety, so we have:

$$\Gamma \vdash_{\mathrm{d}} o[p\{\!|{}^{o'_1}/_{o_1}|\!\}]\{^{\Phi \cup \{o'_1:C\}}_{\Delta \cup \{o'_1 = \emptyset\}}$$

as required.

Attribute Access

Now we consider Attribute Access. This axiom has the form:

$$\Gamma \ \rhd \ o[a?r\ p]\{^{\Phi}_{\Delta} \ \longrightarrow \ p\{\!|{}^{v}/_{r}|\!\}\{^{\Phi}_{\Delta}$$

where $v = \Delta(o)(a)$. The premise of the theorem gives us $\Gamma \vdash_{\mathrm{d}} o[a?r\ p]\{^{\Phi}_{\Delta}$ so we have $\Gamma, \Phi \vdash_{\mathrm{d}} o[a?r\ p]$ and $\Gamma, \Phi \vdash_{\mathrm{d}} \Delta$. From the first statement, we can infer $\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} a?r\ p$. By the rules of the dynamic type system, this implies $\Gamma, \Phi \cup Att_o \ \vdash \ a{:}\texttt{Attr}\{T\}$ and $\Gamma, \Phi \cup Att_o \cup \{r'{:}T\} \vdash_{\mathrm{d}} p\{\!|{}^{r'}/_{r}|\!\}$. Since the state is dynamically type safe, we have $\Gamma, \Phi \cup Att_o \cup \{r'{:}T\} \ \vdash \ v{:}T$. Then by Lemma 6.4, case 3 and case 2, $\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} p\{\!|{}^{v}/_{r}|\!\}$. So $\Gamma, \Phi \vdash_{\mathrm{d}} o[p\{\!|{}^{v}/_{r}|\!\}]$. Thus $\Gamma \vdash_{\mathrm{d}} o[p\{\!|{}^{v}/_{r}|\!\}]\{^{\Phi}_{\Delta}$ as required

Attribute Update

Next, Attribute Update which has the form:

$$\Gamma \ \rhd \ o[a!v\ p]\{^{\Phi}_{\Delta} \ \longrightarrow \ o[p]\{^{\Phi}_{\Delta'}$$

where $o \in \mathrm{Dom}(\Delta)$, and $\Delta'(o_1)(a_1) = \Delta(o_1)(a_1)$ if $o_1 \neq o$ or $a_1 \neq a$ and v otherwise. The premise of the theorem gives us: $\Gamma \vdash_{\mathrm{d}} o[a!v\ p]\{^{\Phi}_{\Delta}$ from which

we deduce $\Gamma, \Phi \vdash_{\mathrm{d}} o[a!v\ p]$ and $\Gamma, \Phi \vdash_{\mathrm{d}} \Delta$. From the first of these we get $\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} a!v\ p$. By the rules of the dynamic typing system, this gives us $\Gamma, \Phi \cup Att_o \vdash v{:}T$, $\Gamma, \Phi \cup Att_o \vdash a{:}\mathtt{Attr}\{T\}$ and $\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} p$. This last point gives us $\Gamma, \Phi \vdash_{\mathrm{d}} o[p]$. Considering state, we can see that $\Gamma, \Phi \vdash_{\mathrm{d}} \Delta'$, since v is appropriately typed. This, with the above dynamic agent type safety statement, gives us $\Gamma \vdash_{\mathrm{d}} o[p]\{^{\Phi}_{\Delta'}$ As required.

Typecase

Lastly, we consider Typecase. This axiom has the form:

$$\Gamma \quad \rhd \quad o[\mathtt{typecase}\ v{:}T\ \{p_0\}\ \mathtt{else}\ \{p_1\}] \quad \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right.$$

$$\longrightarrow \quad o[p_i] \qquad \left\{ \begin{array}{l} \Phi \\ \Delta \end{array} \right.$$

By the main premise of the theorem $\Gamma \vdash_{\mathrm{d}} o[\mathtt{typecase}\ v{:}T\ \{p_0\}\ \mathtt{else}\ \{p_1\}]\{^{\Phi}_{\Delta}$. This gives us $\Gamma, \Phi \vdash_{\mathrm{d}} o[\mathtt{typecase}\ v{:}T\ \{p_0\}\ \mathtt{else}\ \{p_1\}]$ and thus

$$\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} \mathtt{typecase}\ v{:}T\ \{p_0\}\ \mathtt{else}\ \{p_1\}$$

Now by the dynamic typing system rules, we know that $\Phi = \Phi' \cup \{v{:}T'\}$, for some Φ', and from the above statement, we have both $\Gamma, \Phi' \cup Att_o \cup \{v{:}T\} \vdash_{\mathrm{d}} p_0$ and $\Gamma, \Phi' \cup Att_o \cup \{v{:}T'\} \vdash_{\mathrm{d}} p_1$.

Say that $\Gamma, \Phi \vdash v{:}T$. Then $\Gamma \vdash T' \leq T$, and we can use Lemma 6.4, case 5, to weaken the first of the above two statements to give $\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} p_0$ from which I can deduce $\Gamma, \Phi \vdash_{\mathrm{d}} o[p_0]$. In the other case, we have $\Gamma, \Phi \cup Att_o \vdash_{\mathrm{d}} p_1$ anyway, from which we deduce $\Gamma, \Phi \vdash_{\mathrm{d}} o[p_1]$. The state is unchanged by this axiom, so which ever case applies (call it i), we have $\Gamma \vdash_{\mathrm{d}} o[p_i]\{^{\Phi}_{\Delta}$ as required. That was the last of the axioms, so we're done. ■

Theorem 6.7 ***(Type Safety)*** *If* $\Phi_0 \vdash \Gamma$ *and* $\Gamma \rhd g_\Gamma\{^{\Phi_0}_{\emptyset} \implies g\{^{\Phi}_{\Delta}$ *then* $\Gamma \rhd g\{^{\Phi}_{\Delta}$ *is not type violating.*

Proof: This follows from Lemma 6.3 and Theorem 6.6, using an obvious inductive argument on the derivation of $\Gamma \rhd g_\Gamma\{^{\Phi_0}_{\emptyset} \implies g\{^{\Phi}_{\Delta}$ ■

7 Comparison with Other Work

We have built our type system using recursive types and upon a π-calculus core, so we have taken the type system of [PS93] as a basis for our work.

This presents a type system for the π-calculus with recursive channel types and which is statically type checkable. Thus this type system is similar to the subset of ours where primitive types and interface types are dropped. Our proof of the termination, soundness and completeness of our subtyping system is based directly upon their technique.

For a formal definition of trees we choose an approach based on the definitions in [AC91], which considers recursive function types. This paper also contains a counterpart to our expansion function, $\mathrm{E}_{\Gamma}^{\emptyset}(t)$. A system of regular equations, like our set of definitions, can be converted into recursive types. This process, like ours, is validated with respect to trees.

While Oompa, with concurrency and channels, has quite different semantics, the type systems of the various object calculi considered by Abadi and Cardelli in [AC96] can be related to the one we are using. In particular, the type system of the language $\mathbf{Ob}_{1<:\mu}$ with typecase seems similar, as it uses recursion to bind object self-references. Their choice of using object-based languages where methods can be updated increases the expressive power of their language, but has the side effect that some seemingly desirable subtype relationships cannot be established. Within their system methods can be made non-writable, and then (like Oompa) these subtype relationships can be derived. Although predominantly object-based, class structures for some of their languages are given. However, class and interface definitions are used informally, and no equivalent notion to our expansion is described. This means there is no explicit way of dealing with inter-referential definitions in the context of subtyping.

Another system which bears similarity to Oompa is TyCO (Typed Concurrent Objects) [Vas94]. Objects are primitive in TyCO, but class-like entities may be defined using a "let" construction. A corresponding notion of subclassing can similarly be defined. TyCO doesn't have an explicit subtyping system, but a similar effect is achieved by making the rule which type checks invocation apply to any object which has an appropriate method of the correct arity. Thus an object of one type may stand for an object of another as the receiver of an invocation.

A number of approaches to the typing and subtyping of object systems are considered in the literature. Four approaches are compared in [BCP97]. One uses recursion to deal with interface types and is similar to our approach to typing objects. The other approaches use existential types or a combination of existential and recursive types.

The approach to subtyping object languages using existential types to bind object self-references rather than recursive types, is illustrated in [PT92]. The advantage is put forward as a simplification of the underlying theory. In fact, the existential quantifier can be encoded using a universal quantifier,

which seems an obvious simplification in languages with polymorphism where the universal quantifier is already present. Due to the presence of recursive channel types and the absence of this kind of polymorphism in our system, however, it seems desirable to use the recursive quantifiers for dealing with self-references. Abadi and Cardelli [AC96] use a combination of recursive and existential operators to give what they call the *self quantifier*. This has interesting subclassing properties.

8 Conclusions and Future Work

This technical report has given a formal definition of our Oompa calculus. We have provided its syntax and its operational semantics, and given it a type system which can statically check a set of definitions for type correctness. We have presented a technique for converting inter-referential interface types into recursively bound closed types, and our subtyping algorithm is terminating upon these. We used a form of subject reduction to show that our type system is sound, that is, a type checked definition set never leads to an agent which attempts a type violation.

The next step in the development of Oompa will be to develop support for reasoning. In process algebras, the usual approach is to define bisimulations, but Oompa objects have internal state which should make the defining of an appropriate bisimulation particularly interesting. This reasoning is also relevant to the issue of object substitutability, which is very important in modern distributed object systems, where objects encounter each other at run time. A good example is the CORBAservices specification [Obj97], which describes a Trader service which attempts to match suitable CORBA objects to requests. Any implementation of such a trader will need to have some notion of substitutability.

As defined in this report, Oompa is a very simple calculus, and is not intended to be a full programming language. However, there is much that could be added to increase its usability in this regard. Subclassing could easily be added, although the usual problems will need to be dealt with (e.g. name clashes). A syntax for expressions, e.g. $v + 5$, would make the design of Oompa "programs" much easier. Currently, this sort of calculation must be done at either the π-calculus channel level, or the object interface level.

A Free Variables and Substitution

In this appendix, we provide the definitions for substitution on code and types and functions for finding their free variables.

The Set of Free Variables of Code

$$
\begin{array}{rcl}
FV(\texttt{end}) & = & \emptyset \\
FV(\texttt{fork}\{p_0\}\ p_1) & = & FV(p_0) \cup FV(p_1) \\
FV(\texttt{new}\ c\colon \mathrm{ChT}\ p_0) & = & FV(p_0) - \{c\} \\
FV(c!\langle v_1, \ldots v_n\rangle\ p_0) & = & FV(p_0) \cup \{c, v_1, \ldots v_n\} \\
FV(c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_0) & = & (FV(p_0) - \{r_1, \ldots r_n\}) \cup \{c\} \\
FV(o.m!\langle v_1, \ldots v_n\rangle ?(r_1, \ldots r_m)\ p_0) & = & (FV(p_0) - \{r_1, \ldots r_m\}) \\
 & & \quad \cup \{o, v_1, \ldots v_n\} \\
FV(\texttt{create}\ o\colon \mathrm{ClT}\ p_0) & = & FV(p_0) - \{o\} \\
FV(a?r\ p_0) & = & FV(p_0) - \{r\} \\
FV(a!v\ p_0) & = & FV(p_0) \cup \{v\} \\
FV(\texttt{typecase}\ v\colon T\ \{p_0\}\ \texttt{else}\ \{p_1\}) & = & \{v\} \cup FV(p_0) \cup FV(p_1)
\end{array}
$$

Substitution on Code

$$
\begin{array}{rcl}
\texttt{end}\ \rho & = & \texttt{end} \\
\texttt{fork}\{p_0\}\ p_1\ \rho & = & \texttt{fork}\{(p_0\ \rho)\}\ (p_1\ \rho) \\
(\texttt{new}\ c\colon \mathrm{ChT}\ p_0)\ \rho & = & \texttt{new}\ c\colon \mathrm{ChT}\ (p_0\ \rho') \\
 & & \quad \text{where } \rho' = \rho - \{c\}. \\
(c!\langle v_1, \ldots v_n\rangle\ p_0)\ \rho & = & (c\ \rho)!\langle (v_1\ \rho), \ldots (v_n\ \rho)\rangle\ (p_0\ \rho) \\
(c?(r_1\colon T_1, \ldots r_n\colon T_n)\ p_0)\ \rho & = & (c\ \rho)?(r_1\colon T_1, \ldots r_n\colon T_n)\ (p_0\ \rho') \\
 & & \quad \text{where } \rho' = \rho - \{r_1, \ldots r_n\}. \\
(o.m!\langle v_1 \ldots v_n\rangle ?(r_1 \ldots r_m)\ p_0)\ \rho & = & (o\ \rho).m!\langle (v_1\ \rho) \ldots (v_n\ \rho)\rangle ?(r_1 \ldots r_m) \\
 & & \quad (p_0\ \rho') \\
 & & \quad \text{where } \rho' = \rho - \{r_1, \ldots r_m\}. \\
(\texttt{create}\ o\colon \mathrm{ClT}\ p_0)\ \rho & = & \texttt{create}\ o\colon \mathrm{ClT}\ (p_0\ \rho') \\
 & & \quad \text{where } \rho' = \rho - \{o\}. \\
(a?r\ p_0)\ \rho & = & a?r\ (p_0\ \rho') \\
 & & \quad \text{where } \rho' = \rho - \{r\}. \\
(a!v\ p_0)\ \rho & = & a!(v\ \rho)\ (p_0\ \rho) \\
(\texttt{typecase}\ v\colon T\ \{p_0\}\ \texttt{else}\ \{p_1\})\rho & = & \texttt{typecase}\ (v\rho)\colon T\ \{p_0\rho\}\ \texttt{else}\ \{p_1\rho\}
\end{array}
$$

The Set of Free Types Variables of a Type

$$
\begin{array}{rcl}
FTV(\texttt{Long}) & = & \emptyset \\
FTV(\texttt{Char}) & = & \emptyset \\
FTV(\texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\}) & = & FTV(\mathrm{SgT}_1) \cup \ldots FTV(\mathrm{SgT}_n) \\
FTV(m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle) & = & FTV(T_1) \cup \ldots FTV(T_n) \\
 & & \cup FTV(T'_1) \cup \ldots FTV(T'_{n'}) \\
FTV(\texttt{Chan}\langle T_1, \ldots T_n\rangle) & = & FTV(T_1) \ldots FTV(T_n) \\
FTV(\texttt{InCh}\langle T_1, \ldots T_n\rangle) & = & FTV(T_1) \ldots FTV(T_n) \\
FTV(\texttt{OuCh}\langle T_1, \ldots T_n\rangle) & = & FTV(T_1) \ldots FTV(T_n) \\
FTV(\texttt{rec}\ t.T) & = & FTV(T) - \{t\} \\
FTV(t) & = & \{t\}
\end{array}
$$

Substitution on Types

$$
\begin{array}{rcl}
\texttt{Long}\ \rho & = & \texttt{Long} \\
\texttt{Char}\ \rho & = & \texttt{Char} \\
\texttt{Intf}\{\mathrm{SgT}_1, \ldots \mathrm{SgT}_n\}\ \rho & = & \texttt{Intf}\{(\mathrm{SgT}_1\ \rho), \ldots (\mathrm{SgT}_n\ \rho)\} \\
m?(T_1, \ldots T_n)!\langle T'_1, \ldots T'_{n'}\rangle\ \rho & = & m?((T_1\ \rho), \ldots (T_n\ \rho))!\langle (T'_1\ \rho), \ldots (T'_{n'}\ \rho)\rangle \\
\texttt{Chan}\langle T_1, \ldots T_n\rangle\ \rho & = & \texttt{Chan}\langle (T_1\ \rho), \ldots (T_n\ \rho)\rangle \\
\texttt{InCh}\langle T_1, \ldots T_n\rangle\ \rho & = & \texttt{InCh}\langle (T_1\ \rho), \ldots (T_n\ \rho)\rangle \\
\texttt{OuCh}\langle T_1, \ldots T_n\rangle\ \rho & = & \texttt{OuCh}\langle (T_1\ \rho), \ldots (T_n\ \rho)\rangle
\end{array}
$$

In the case of recursive types, we assume that α-substitution is used to ensure that variable capture does not occur. We don't consider the issues here.

$$(\texttt{rec}\ t.T)\ \rho = \texttt{rec}\ t'.(T\{\!|^{t'}/_{t}|\!\}\ \rho)$$

References

[AC91] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 104–118, Orlando, FL, USA, January 1991. ACM Press.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[BCP97] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, September 1997. An earlier version was presented as an invited lecture at the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[Obj97] The Object Management Group. *CORBAservices: Common Object Services Specification*, November 1997.

[OMG98] OMG. *CORBA 2.2 Specification*, February 1998.

[PS93] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 376–385, Montreal, Canada, 19–23 June 1993. IEEE Computer Society Press.

[PT92] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. Technical report, Department of Computer Science, University of Edinburgh, Edinburgh, U.K., August 1992.

[San96] Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. Technical Report RR-3000, Inria, Institut National de Recherche en Informatique et en Automatique, 1996.

[TBD00] Malcolm Tyrrell, Andrew Butterfield, and Alexis Donnelly. Oo-motivated process algebra: A calculus for corba-like systems. In *Third Workshop in Rigorous Object-Oriented Methods, York, England*, January 2000.

[Vas94] V. T. Vasconcelos. Typed concurrent objects. *Lecture Notes in Computer Science*, 821, 1994.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Department of Computer Science, Rice University, Houston, Texas, April 1991.