Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Masters's Thesis

Mechanized Formalization of a Transformation from an Extensible Spi Calculus to Java

submitted by

Alex Busenius

on 04.04.2011

Supervisor

Prof. Dr. Michael Backes

Advisor

Cătălin Hriţcu

Reviewers

Prof. Dr. Michael Backes
Matteo Maffei, Ph.D

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 04.04.2011 _____
(Datum / Date)

_____
(Unterschrift / Signature)

Abstract

This thesis presents a formalization of a transformation of protocols modeled in a variant of Spi calculus [AB05] to a substantial subset of Java. We formalize the transformation in the Coq proof assistant [Tea10]. Our source language is designed to be easily extensible and flexible enough to allow specification of complex real-life protocols such as TLS. The target language combines a formalization of Java called Jinja with Threads [Loc08], with Variant Parametric Featherweight Java [IV06]. It features concurrency with shared memory and synchronization, exception handling and a sophisticated type system with parametric polymorphism. We prove that the programs generated by the transformation of well-typed models are well-typed.

Acknowledgments

I am deeply grateful to my advisor, Cătălin Hriţcu, for his continuous support and endless enthusiasm that kept me motivated during all this time. He could always help me to find the right solution for the problems I encountered.

I would also like to thank Jan Schwinghammer and Christian Doczkal, who helped me to learn Coq, and Andreas Lochbihler for his explanations regarding Jinja with Threads.

Contents

1. Introduction

1.1. Motivation

Cryptographic protocols play a very important role in our everyday life. In the era of smartphones and mobile Internet we constantly use a myriad of different protocols to communicate with each other, to do online banking or even to vote for the next government. Many great cryptographers and security researchers invested a lot of energy into designing secure cryptographic protocols that make all these applications possible. However, the experience shows that many problems arise not because of the broken design, but because of the human mistakes made in the implementation of the protocols.

There are several approaches to ensure that the implementation of a cryptographic protocol is correct and secure. One of them is to formally model the protocol in an abstract calculus, such as the Spi calculus [AG99], prove the desired security properties and generate the implementation for the protocol in some widely used programming language such as Java. There are various verification tools that can automatically prove many interesting security properties of protocol models such as secrecy and authentication. One of the most well-known tools that can verify protocols modeled in Spi calculus is ProVerif [Bla01]. Furthermore, these models can be analyzed by many type systems [Aba99, AB01, GJ04, HJ06, BFM07, BCFM07, FGM07, BHM08] that statically enforce some of the important security properties.

The generation of the implementation can also be automated. The existing code generation tools usually use the verified protocol model with some additional specification that defines how to implement the protocol messages in an interoperable way and generate the protocol implementation in some mainstream programming language. There are quite a few experimental tools for automatic code generation that can be used to generate code for complex cryptographic protocols, for example CIL2Java by J. Millen and F. Muller [MM01], the Sprite tool by B. Tobler [Tob05], Spi2java by A. Pironti, R. Sisto, L. Durante and D. Pozza [PS07] and expi2java by A. Busenius [Bus08].

In the author's bachelor's thesis [Bus08], we started going in this direction by developing expi2java, a flexible and extensible code generation tool that is powerful enough to generate an interoperable implementation of complex real-life protocols such as TLS. We use a variant of the Spi calculus we call the Extensible Spi Calculus as the input language to model the protocols and generate implementations in Java. We use an expressive type system with parametric types and configurations to specify the low-level format of the messages sent in the protocols in a flexible way. This allows using ProVerif to prove the protocol model secure and ensures that the generated code can easily be integrated into existing applications. However, like many other code generation tools, expi2java lacks a formal model of the transformation between the source and the target language that would allow to prove that the transformation

is correct and secure. Without the formal proof, we only have a slightly better confidence that the automated code generation will generate a better code than a manual implementation.

In order to prove the transformation secure, we need to develop a formalization of the source and the target language, formalize the transformation between them, define what security properties we want to have and finally prove that the transformation preserves them. Unsurprisingly, this task poses quite a few challenges. First of all, the formalization of the Extensible Spi Calculus, our source language, must include a way to represent the low-level configurations in a flexible way that reflects the customizability of expi2java. The model of Java must be very faithful and support all the features we need to implement the protocols, such as concurrency, synchronization between threads and exception handling. These features, especially the concurrency, make the formalization much more complex and hard to work with. Because of this complexity it is hard to find a simple way to relate both languages that would allow us to prove the desired properties.

The complexity of this task significantly increases the risk of making mistakes in the formalization or the proof that would render the benefits of having a proved transformation useless. A good way to avoid the mistakes is to use a proof assistant, such as Coq or Isabelle/HOL, to define a mechanized formalization of the transformation and to machine-check the proofs. Using an interactive proof assistant has also other benefits. It allows to extend the formalization in an iterative way and makes sure that all the proofs and definitions are adapted to work with those extensions.

Our long-time goal is to write a machine-checked proof for the correctness and preservation of security properties by the transformation used in expi2java. This would make expi2java the first automatic code generator that uses a machine-checked provably secure transformation to generate interoperable implementations for cryptographic protocols.

1.2. Contributions

In this thesis we formalize the transformation from the Extensible Spi Calculus to a substantial subset of Java in the Coq proof assistant [Tea10] and prove that the transformation is well-typed.

The mechanized formalization of Extensible Spi Calculus is based on the definitions from the author's bachelor's thesis [Bus08]. We have adapted these definitions for use with Coq and formally defined the low-level configurations used to specify the implementations details for each data type.

We formalize the target language based on one of the existing formalizations of Java, called Jinja with Threads [Loc08]. We extend Jinja with Threads with a sophisticated type system based on the Variant Parametric Types [IV06] to support variant subtyping needed to implement the corresponding features of the type system for the Extensible Spi Calculus. We call the resulting language Variant Parametric Jinja (VPJ).

The transformation is formalized in two steps. First we transform models written in the Extensible Spi Calculus into the Global Expi Calculus, an intermediate language that has

a simpler semantics, and then transform models in this intermediate language to VPJ programs. The transformation includes a VPJ implementation of structural equality and pattern matching on terms, needed to model the semantics of Expi processes in an object-oriented language. Additionally, we have implemented a symbolic library that abstracts away cryptographic primitives as symbolic terms and networking as pi-calculus-like channels. We prove that the symbolic library is well-typed.

Furthermore, we prove that the transformation generates well-typed programs when starting from well typed models. This is an important step towards our longer-term goal of proving the correctness of the transformation. This also helped a lot during the formalization phase to spot problems and make sure the various definitions fit well together.

We have also significantly extended the implementation of this transformation started as a part of the author's bachelor's thesis to the point where it can be used in the real-world scenarios. The expi2java tool that implements our transformation now supports inferring type parameters, allows for a much compacter representation of complex protocols, supports more cryptographic primitives by default and has many other improvements that make it not only usable but also really useful.

A case study on automatically generating the client and server sides for the Transport Layer Security (TLS) protocol [DA99] indicates the progress we made with expi2java since the first version presented in the author's bachelor's thesis. The generated implementation supports dynamic selection of the cipher used to establish an encrypted connection and is fully interoperable with other standard-compliant implementations.

1.3. Outline

Chapter 2 describes the formalization of our source language, the Extensible Spi Calculus. In Chapter 3 we present the formalization of the target language, Variant Parametric Jinja and the corresponding type system with variant parametric types. In Chapter 4 we define the transformation and present the idea behind our proof showing that the programs generated by our transformation are well-typed. Chapter 5 describes the implementation of these ideas in the expi2java tool on the example of the TLS protocol. In Chapter 6 we give a short overview of the related work. Finally, in Chapter 7 we again summarize our contributions, discuss directions for future work and conclude.

2. Formalizing Extensible Spi Calculus

Our formalization consists of three main parts: the source language (the Extensible Spi Calculus), the target language (a variant parametric variant of Jinja, which we will call VPJ) and the actual transformation.

The source language is a variant of the Spi calculus, a process calculus for modeling cryptographic protocols. This calculus is essentially the one by M. Abadi and B. Blanchet [AB05], but we extend it with configurations (Section 2.3) and define a type system for it (Section 2.5). In this section we define the syntax, operational semantics, and type system of this calculus. We use the syntax of the Coq proof assistant [Tea10] in our presentation; the listed fragments are extracted directly from our actual formalization.

2.1. Abstract Syntax

2.1.1. Terms, Names and Constructors

In the Extensible Spi Calculus, *terms* are used to model data and *processes* are used to model the behavior of the protocol participants and the communication between them. The set of terms contains *Expi names* (which represent constant data), *variables* and *constructor applications*.

```
Section ParametrizedByName.

(* Contains identifiers used as names of types, constructors and destructors *)
Variable Idents : Set.

(* Configuration name is a string used to distinguish different implementations
   of types, constructors and destructors *)
Definition cfg_name := string.

Inductive nam : Set :=
 | Nam_b : nat → nam
 | Nam_f : atom → nam.

Inductive term : Set :=
 | term_Nam : nam → term
 | term_Var_b : nat → term
 | term_Var_f : atom → term
 | term_Ctor : cfg_name → Idents → list type → list term → term.
```

```
End ParametrizedByName.

Definition  Vf Id a :  @term Id := term_Var_f a.
Definition  Vb Id n :  @term Id := term_Var_b n.
Definition  Nf Id a :  @term Id := term_Nam (Nam_f a).
Definition  Nb Id n :  @term Id := term_Nam (Nam_b n).
```

Listing 1: Expi Names and Terms

We use *locally-nameless* representation [Gor93, ACP+08] for Expi names and variables to avoid the issues related to α-conversion. In this representation *free* variables are represented using their (variable) names (see `Nam_f` and `term_Var_f`) and *bound* variables are represented using de Bruijn indices [dB72] referring to their binders (see `Nam_b` and `term_Var_b`). We use `nat` (natural numbers) for de Bruijn indices and `atom` (an abstract data type defined in the metatheory library [ACP+08]) for variable names.

An Expi term can either be an Expi name (`nam`), a variable (`term_Var_f` or `term_Var_b`) or a constructor application (`term_Ctor`). Since the set of constructors is not fixed and can be extended, we parameterize `term_Ctor` with a configuration name `cfg_name` and a constructor identifier contained in the set `Idents`. We explain how exactly we define the set of constructors using configurations in Section 2.3. Constructors can have a parametric type, in which case the constructor application must provide a list of type annotations used to instantiate the constructor type. We explain this in more detail in Section 2.5. In this section we will only use constructors with a simple non-parametric type and therefore `nil` as type annotation.

We define Coq notations for Expi names and variables to improve readability. The notation `Nf a` stands for a free name called `a`, while `Vf a` stands for a free variable called `a`. The notation `Nb n` stands for a bound name with de Bruijn index `n`, while `Vb n` stands for a bound variable with de Bruijn index `n`. Bound variables and bound names are counted separately.

2.1.2. Destructors

Destructors are partial functions that can be applied to (lists of) terms. Similar to constructors, we parameterize destructors with a configuration name and a destructor identifier. They can also have a parametric type and therefore take a list of type annotations as argument.

```
Section ParametrizedByName.

Inductive  dtor : Set :=
 |  dtor_Dtor : cfg_name → Idents → list type → list term → dtor.

End ParametrizedByName.
```

Listing 2: Expi Destructors

The semantics of destructors is defined by the reduction relation `g_red`: they can either succeed and provide some term as the result or fail. The reduction relation is not fixed, it is part of the configuration. We explain the configurations in more detail in Section 2.3.

2.1.3. Processes

The *processes* are used to model the behavior of protocol participants and the communication between them [AB05]. A specific characteristic of our calculus is that the replication process can only be followed by the input process, as in [FGM07, BHM08, BGHM09]. This is the most common way to use replication in specifications, and at the same time it is reasonably easy to implement such a usage of replication as a server that waits for requests on a channel and spawns a new thread to processes each incoming message.

```
Section ParametrizedByName.

Inductive  proc : Set :=
 |  proc_out : term → term → proc → proc
 |  proc_in : term → proc → proc
 |  proc_bangin : term → proc → proc
 |  proc_let : dtor → proc → proc → proc
 |  proc_new : type → proc → proc
 |  proc_fork : proc → proc → proc
 |  proc_null : proc.

End ParametrizedByName.

(* Binds the message received on channel c in variable x in process P *)
Notation  ":in( c , x );; P" := (proc_in c (close_proc_wrt_term x P))
                                (at level 60) :  proc_scope.
(* Binds the message received on channel c in variable x in process P *)
Notation  "!in( c , x );; P" := (proc_bangin c (close_proc_wrt_term x P))
                                (at level 60) :  proc_scope.
(* Binds the result of a successful application of g in variable x in process P *)
Notation  ":let x ::= g :in P :else Q" := (proc_let g (close_proc_wrt_term x P) Q)
                                          (at level 60) :  proc_scope.
(* Binds a fresh Expi name a of type T in process P *)
Notation  ":new a :::  T ;; P" := (@proc_new _ T (close_proc_wrt_nam a P))
                                  (at level 60) :  proc_scope.

Notation  ":out( c , t );; P" := (proc_out c t P) (at level 60) :  proc_scope.
Notation  "P :| Q" := (proc_fork P Q) (at level 60) :  proc_scope.
Notation  ":0" := (proc_null) (at level 60) :  proc_scope.
```

Listing 3: Expi Processes

The syntax of terms and processes in locally-nameless representation allows to define "invalid" processes. For example, the process `proc_new T (proc_out (Nb 0) (Nb 1) proc_null)` is invalid, because it contains bound name `Nb 1` with de Bruijn index 1. Since there is only one binding process `proc_new` in scope, all de Bruijn indices must be < 1. We need to ensure that all processes, destructors and terms are *locally-closed* to exclude such "invalid" processes from being used. A syntactic construct is called locally-closed when it contains no dangling de Bruijn indices. This property can be checked using the generated propositions `lc_nam`, `lc_term`, `lc_dtor` and `lc_proc` (see Appendix A.1.2).

We define Coq notations for all processes in order to make definitions of larger processes more readable. The notations look similar to the usual way these processes are defined in literature, except that the restriction process `proc_new` additionally takes a type parameter. Note that we use the explicit Expi name or variable parameter in cases when the corresponding process binds it (i.e., `proc_in`, `proc_bangin`, `proc_let` and `proc_new`) just to improve readability. The notations "close" the continuation process with respect to this free name, effectively replacing uses of this free name with the corresponding de Bruijn indices.

2.2. A Toy Example

```
Section Example1.

Variable X : Set.   (* Set of identifiers, not important for now *)
Variable T : @type X.  (* The exact type is not important for now *)
Variable c : atom.   (* The name of the channel *)
Variable n : atom.   (* The name of the nonce *)

Definition process_a : @proc X :=
  :new n ::: T ;;  (* Generate name n *)
  :out(Vf c, Vf n);;  (* Send n over c *)
  :0.

Definition process_b : @proc X :=
  :in(Vf c, n);;  (* Receive n over c *)
  :0.

Definition process_main :=
  :new c ::: (type_Channel "TLS" T) ;;  (* Generate channel c *)
  process_a :| process_b.

End Example1.
```

Listing 4: A Simple Example

The main process starts by generating a fresh Expi name `c` that is used as a channel later on. Then it starts the two participant processes `process_a` and `process_b` in parallel. Participant A generates a fresh Expi name `n`, then sends it over the channel `c`. Participant B just receives an Expi name over the channel `c`.

Note that the type annotation for the restriction processes is left abstract. The types are introduced later in Section 2.5.

2.3. Configurations

An important feature of the Extensible Spi Calculus is the customizability. The user can extend the sets of types, constructors and destructors, redefine the reduction relation of

destructors and provide different implementations for cryptographic primitives. We formalize these features using what we call a *configuration* – three sets of identifiers (`Idents`) and several functions that define the behavior of types, constructors and destructors. These predicates can be used in any later definitions that take a configuration as a parameter. For example, the reduction relation on processes defined in Section 2.4 uses the destructor reduction relation `g_red` defined in the configuration.

2.3.1. Definition

```
Section ParametrizedByName.

(* The configuration is implicitly parameterized by the set of identifiers Idents.  *)

Record  config : Type :=
 Cfg {
  t_idents           : list Idents;
  f_idents           : list Idents;
  g_idents           : list Idents;
  t_varmap           : Idents → option (list variance);
  is_t_gen           : type → bool;
  f_type             : cfg_name → Idents → option fun_type;
  g_type             : cfg_name → Idents → option fun_type;
  g_rules            : cfg_name → Idents → list (nat × list term × term);
  g_red              : dtor → option term;
  (* Consistency assumptions *)
  eq_Idents_dec      : ∀ (x y : Idents), {x = y} + {x ≠ y};
  distinct_names     : ∀ x, {In x t_idents} + {In x f_idents} + {In x g_idents};
  t_varmap_some      : ∀ x, In x t_idents → t_varmap x ≠ None;
  t_varmap_none      : ∀ x, ¬ In x t_idents → t_varmap x = None;
  is_t_gen_top       : is_t_gen type_Top = false;
  is_t_gen_var       : ∀ x, is_t_gen (type_Var x) = false;
  is_t_gen_channel   : ∀ c t, is_t_gen (type_Channel c t) = true;
  is_t_gen_false     : ∀ c x ts, ¬ In x t_idents →
                                 is_t_gen (type_Nested c x ts) = false;
  f_type_some        : ∀ c x, In x f_idents → f_type c x ≠ None;
  f_type_none        : ∀ c x, ¬ In x f_idents → f_type c x = None;
  g_type_some        : ∀ c x, In x g_idents → g_type c x ≠ None;
  g_type_none        : ∀ c x, ¬ In x g_idents → g_type c x = None;
  g_rules_list       : ∀ c x, In x g_idents → g_rules c x ≠ nil;
  g_rules_nil        : ∀ c x, ¬ In x g_idents → g_rules c x = nil;
  g_red_reduces      : ∀ c x n args ret annos,
                       In (n, args, ret) (g_rules c x) →
                       g_red (dtor_Dtor c x annos args) = Some ret
}.

End ParametrizedByName.
```

Listing 5: Configuration

The following configuration parameters are used to fine-tune the Extensible Spi Calculus:

`t_idents` - a list of type identifiers
`f_idents` - a list of constructor identifiers
`g_idents` - a list of destructor identifiers
`t_varmap` - a partial function defining the variance of all types (see Section 2.5)
`is_t_gen` - a boolean function that returns true if a type is generative
`f_type` - a partial function that defines the functional type (i.e., the type of all arguments and the result type, see Section 2.5 for more details) of each constructor
`g_type` - a partial function that defines the functional type of each destructor
`g_rules` - a function that returns the reduction rules for each destructor, where a reduction rule is a triple consisting of the number of variables, a list of terms to match the arguments and the resulting term
`g_red` - a partial function that applies the corresponding reduction rule for each destructor and returns a term if and only if the given destructor application succeeds

In order to use the configuration we need to ensure that it is *consistent*. The consistency is defined as a set of assumptions that need to be proved and provided as a part of the configuration. Note that since Coq is based on the Curry-Howard correspondence we can treat the proofs and terms in the same way and conveniently put both into the same record. The consistency assumptions are:

`eq_Idents_dec` - equality on the set of identifiers `Idents` is decidable
`distinct_names` - the set of identifiers `Idents` can be split into 3 distinct sets of type, constructor and destructor names
`t_varmap_some` - function `t_varmap` is defined for all types
`t_varmap_none` - function `t_varmap` is not defined for non-types
`is_t_gen_top` - the Top type is not generative
`is_t_gen_var` - type variables are not generative
`is_t_gen_channel` - the Channel type is generative
`is_t_gen_false` - `is_t_gen` returns false for non-types
`f_type_some` - function `f_type` is defined for all constructors
`f_type_none` - function `f_type` is not defined for non-constructors
`g_type_some` - function `g_type` is defined for all destructors
`g_type_none` - function `g_type` is not defined for non-destructors
`g_rules_list` - `g_rules` returns a not empty list for all destructors
`g_rules_nil` - `g_rules` returns nil for non-destructors
`g_red_reduces` - `g_rules` and `g_red` agree on their results

Additionally, one must prove that all destructors are consistent. The destructor consistency assumption is not part of the configuration, because it would cause a circular dependency (the definition of destructor consistency uses the term typing relation, which needs configuration). Destructor consistency is an important property, we explain it in more detail in Section 2.8.

2.3.2. Default Configuration

We define a default configuration that contains commonly used cryptographic primitives and the corresponding types such as symmetric encryption and pairs. This configuration is sufficient to translate most of the popular cryptographic protocols such as TLS [Bus08]. We instantiate the set of identifiers `Idents` by `Names`, give the remaining definitions, prove the consistency assumptions (see Section 2.8) and combine them in the configuration record named `config_default`.

```
Inductive  Names : Type :=
 |  TBool |  TInt |  TSymEnc |  TSymKey |  TPair |  THash |  TPubEnc |  TKeyPair |  TPubKey
     |  TPrivKey |  TSigned |  TSigKey |  TVerKey
 |  Cenc |  Ctru |  Cfls |  Czero |  Csucc |  Cpair |  Ch |  Chmac |  Cenca |  Cpk |  Csk
     |  Csign |  Csigk |  Cvk
 |  Ddec |  Deq |  Did |  Dpre |  Dfst |  Dsnd |  Ddeca |  Dmsg |  Dver.

Lemma  eq_names_dec : ∀ x y : Names, {x = y} + {x ≠ y}.

Definition  t_idents_default : list Names :=
  (TBool :: TInt :: TSymEnc :: TSymKey :: TPair :: THash :: TPubEnc :: TKeyPair ::
     TPubKey :: TPrivKey :: TSigned :: TSigKey :: TVerKey :: nil).

Definition  f_idents_default : list Names :=
  (Cenc :: Ctru :: Cfls :: Czero :: Csucc :: Cpair :: Ch :: Chmac :: Cenca ::
     Cpk :: Csk :: Csign :: Csigk :: Cvk :: nil).

Definition  g_idents_default : list Names :=
  (Ddec :: Deq :: Did :: Dpre :: Dfst :: Dsnd :: Ddeca :: Dmsg :: Dver :: nil).

Definition  Var n := @term_Var_b Names n.
Definition  fenc c t m k := term_Ctor c Cenc (t :: nil) (m :: k :: nil).
Definition  ftru c := term_Ctor c Ctru nil nil.
Definition  ffls c := term_Ctor c Cfls nil nil.
Definition  fzero c := term_Ctor c Czero nil nil.
Definition  fsucc c n := term_Ctor c Csucc nil (n :: nil).
Definition  fpair c t u x y := term_Ctor c Cpair (t :: u :: nil) (x :: y :: nil).
Definition  fh c t m := term_Ctor c Ch (t :: nil) (m :: nil).
Definition  fhmac c t m k := term_Ctor c Chmac (t :: nil) (m :: k :: nil).
Definition  fenca c t m k := term_Ctor c Cenca (t :: nil) (m :: k :: nil).
Definition  fpk c t kp := term_Ctor c Cpk (t :: nil) (kp :: nil).
Definition  fsk c t kp := term_Ctor c Csk (t :: nil) (kp :: nil).
Definition  fsign c t m k := term_Ctor c Csign (t :: nil) (m :: k :: nil).
Definition  fsigk c t kp := term_Ctor c Csigk (t :: nil) (kp :: nil).
Definition  fvk c t kp := term_Ctor c Cvk (t :: nil) (kp :: nil).

Definition  gdec c t e k := dtor_Dtor c Ddec (t :: nil) (e :: k :: nil).
Definition  geq c t x y := dtor_Dtor c Deq (t :: nil) (x :: y :: nil).
Definition  gid c t x := dtor_Dtor c Did (t :: nil) (x :: nil).
Definition  gpre c n := dtor_Dtor c Dpre nil (n :: nil).
Definition  gfst c t u p := dtor_Dtor c Dfst (t :: u :: nil) (p :: nil).
Definition  gsnd c t u p := dtor_Dtor c Dsnd (t :: u :: nil) (p :: nil).
Definition  gdeca c t e k := dtor_Dtor c Ddeca (t :: nil) (e :: k :: nil).
```

```
Definition  gmsg c t s := dtor_Dtor c Dmsg (t :: nil) (s :: nil).
Definition  gver c t s k := dtor_Dtor c Dver (t :: nil) (s :: k :: nil).


Lemma  distinct_names_default :
  ∀ x, {In x t_idents_default} + {In x f_idents_default} + {In x g_idents_default}.

Lemma  t_varmap_some_default :
  ∀ x, In x t_idents_default → t_varmap_default x ≠ None.

Lemma  t_varmap_none_default :
  ∀ x, ¬ In x t_idents_default → t_varmap_default x = None.

Lemma  is_t_gen_top_default : is_t_gen_default type_Top = false.

Lemma  is_t_gen_var_default : ∀ x, is_t_gen_default (type_Var x) = false.

Lemma  is_t_gen_channel_default :
  ∀ c t, is_t_gen_default (type_Channel c t) = true.

Lemma  is_t_gen_false_default : ∀ c x ts, ¬ In x t_idents_default →
                                  is_t_gen_default (type_Nested c x ts) = false.

Lemma  f_type_some_default : ∀ c x, In x f_idents_default →
                               f_type_default c x ≠ None.

Lemma  f_type_none_default : ∀ c x, ¬ In x f_idents_default →
                               f_type_default c x = None.

Lemma  g_type_some_default : ∀ c x, In x g_idents_default →
                               g_type_default c x ≠ None.

Lemma  g_type_none_default : ∀ c x, ¬ In x g_idents_default →
                               g_type_default c x = None.

Lemma  g_rules_list_default : ∀ c x, In x g_idents_default →
                                g_rules_default c x ≠ nil.

Lemma  g_rules_nil_default : ∀ c x, ¬ In x g_idents_default →
                               g_rules_default c x = nil.

Lemma  g_red_reduces_default :
  ∀ c x n args ret annos,
    In (n, args, ret) (g_rules_default c x) →
    g_red_default (dtor_Dtor c x annos args) = Some ret.

Definition  config_default : config :=
  Cfg Names eq_names_dec
      t_idents_default f_idents_default g_idents_default
      t_varmap_default is_t_gen_default
      f_type_default g_type_default
      g_rules_default g_red_default
      distinct_names_default
      t_varmap_some_default t_varmap_none_default
      is_t_gen_top_default is_t_gen_var_default
      is_t_gen_channel_default is_t_gen_false_default
      f_type_some_default f_type_none_default
```

```
g_type_some_default g_type_none_default
g_rules_list_default g_rules_nil_default
g_red_reduces_default.
```

Listing 6: Default Configuration

2.4. Operational Semantics

The semantics of the calculus is quite standard and is defined by a structural equivalence relation `pequiv` and an internal reduction relation `red`.

Structural equivalence relates the processes that are considered equivalent up to syntactic rearrangement. The only difference to the usual semantics [AB05] is the absence of the binder swapping rule that allows to rearrange restriction processes `proc_new`. This rule is not important for reduction and is difficult to define in a locally-closed representation.

The definition of `pequiv` uses the proposition `lc_proc` which ensures that the corresponding processes are locally-closed. Another function related to locally-closed representation is `open_proc_wrt_nam` which replaces the bound name with de Bruijn index 0 by the given free Expi name, and shifts all other names down accordingly. The finite set `L` of type `vars` contains all currently used free names. It is used to ensure freshness of each Expi name used to open a process.

```
Section ParametrizedByName.

Inductive  pequiv : proc → proc → Prop :=
| pequiv_null : ∀ (P : proc),
    lc_proc P →
    pequiv (proc_fork P proc_null) P
| pequiv_comm : ∀ (P Q : proc),
    lc_proc Q →
    lc_proc P →
    pequiv (proc_fork P Q) (proc_fork Q P)
| pequiv_assoc : ∀ (P Q R : proc),
    lc_proc P →
    lc_proc Q →
    lc_proc R →
    pequiv (proc_fork (proc_fork P Q) R) (proc_fork P (proc_fork Q R))
| pequiv_scope_extrusion : ∀ (L : vars) (P : proc) (T : type) (Q : proc),
    lc_proc (proc_new T Q) →
    lc_proc (P) →
    pequiv (proc_fork P (proc_new T Q)) (proc_new T (proc_fork P Q))
| pequiv_fork : ∀ (P R Q : proc),
    lc_proc R →
    pequiv P Q →
    pequiv (proc_fork P R) (proc_fork Q R)
| pequiv_fork_new : ∀ (L : vars) (T : type) (P Q : proc),
    (∀ a, a ∉ L →
       pequiv (open_proc_wrt_nam P (Nam_f a))
```

```
              (open_proc_wrt_nam Q (Nam_f a))) →
    pequiv (proc_new T P) (proc_new T Q)
| pequiv_refl : ∀ (P : proc),
    lc_proc P →
    pequiv P P
| pequiv_symm : ∀ (P Q : proc),
    pequiv Q P →
    pequiv P Q
| pequiv_trans : ∀ (P R Q : proc),
    pequiv P Q →
    pequiv Q R →
    pequiv P R.

End ParametrizedByName.
```

Listing 7: Structural Equivalence

Internal reduction defines the semantics of communication and destructor application. This definition is parameterized by a configuration `C` which is needed for the destructor reduction relation `g_red` in cases `red_dtor` and `red_else`.

In addition to the `lc_proc` relation used in `pequiv`, the definition of `red` uses a substitution function `subst_term_in_proc` that replaces a free variable by a given term.

```
Section ParametrizedByName.

Variable  C : config.

Inductive  red : proc → proc → Prop :=
| red_io : ∀ (c : atom) (t : term) (P Q : proc),
    lc_proc (proc_in (term_Nam (Nam_f c)) Q) →
    lc_proc P →
    lc_term t →
    red (proc_fork (proc_out (term_Nam (Nam_f c)) t P)
                   (proc_in (term_Nam (Nam_f c)) Q))
        (proc_fork P (open_proc_wrt_term Q t))
| red_bangio : ∀ (c : atom) (t : term) (P Q : proc),
    lc_proc P →
    lc_term t →
    lc_proc (proc_bangin (term_Nam (Nam_f c)) Q) →
    red (proc_fork (proc_out (term_Nam (Nam_f c)) t P)
                   (proc_bangin (term_Nam (Nam_f c)) Q))
        (proc_fork (proc_fork P (open_proc_wrt_term Q t))
                   (proc_bangin (term_Nam (Nam_f c)) Q))
| red_dtor : ∀ (g : dtor) (P Q : proc) (t : term),
    lc_proc (proc_let g P Q) →
    lc_proc Q →
    g_red C g = Some t →
    lc_term t →
    red (proc_let g P Q) (open_proc_wrt_term P t)
| red_else : ∀ (g : dtor) (P Q : proc),
```

```
    lc_proc (proc_let g P Q) →
    lc_proc Q →
    g_red C g = None →
    red (proc_let g P Q) Q
| red_fork : ∀ (P R Q : proc),
    lc_proc R →
    red P Q →
    red (proc_fork P R) (proc_fork Q R)
| red_new : ∀ (L : vars) (T : type) (P Q : proc),
    (∀ a, a ∉ L →
       red (open_proc_wrt_nam P (Nam_f a)) (open_proc_wrt_nam Q (Nam_f a))) →
    red (proc_new T P) (proc_new T Q)
| red_equiv : ∀ (P Q P' Q' : proc),
    pequiv P P' →
    red P' Q' →
    pequiv Q' Q →
    red P Q.

End ParametrizedByName.
```

Listing 8: Internal Reduction

Our default configuration (see Section 2.3.2) contains a set of destructors commonly used in cryptographic protocols. The semantics of default destructors is defined using the reduction rules mapping `g_rules_default` and default destructor reduction relation `g_red_default`.

```
Definition  g_rules_default (c : cfg_name) (gn : Names)
                          : list (nat × list (@term Names) × @term Names) :=
let x := Var 0 in
let y := Var 1 in
match gn with
 | Ddec ⇒ (2, (fenc c Top x y) :: y :: nil, x) :: nil
 | Deq ⇒ (1, x :: x :: nil, x) :: nil
 | Did ⇒ (1, x :: nil, x) :: nil
 | Dpre ⇒ (1, (fsucc c x) :: nil, x) :: (0, (fzero c) :: nil, fzero c) :: nil
 | Dfst ⇒ (2, (fpair c Top Top x y) :: nil, x) :: nil
 | Dsnd ⇒ (2, (fpair c Top Top x y) :: nil, y) :: nil
 | Ddeca ⇒ (2, (fenca c Top x (fpk c Top y)) :: (fsk c Top y) :: nil, x) :: nil
 | Dmsg ⇒ (2, (fsign c Top x (fsigk c Top y)) :: nil, x) :: nil
 | Dver ⇒ (2, (fsign c Top x (fsigk c Top y)) :: (fvk c Top y) :: nil, x) :: nil
 | _ ⇒ nil
end.

Definition  g_red_default (g : dtor) : option term :=
let (gcfg, gn, ts, xs) := g in
match gn, xs with
 | Ddec, (term_Ctor c Cenc _ (x :: k' :: nil)) :: k :: nil
                 ⇒ if bool_of (TermDec.eq_dec k k')
                       && bool_of (StringDec.eq_dec gcfg c)
                   then Some x else None
 | Deq, (x :: y :: nil)
```

```
                ⇒ if TermDec.eq_dec x y then Some x else None
| Did, (x :: nil) ⇒ Some x
| Dpre, (term_Ctor c Csucc _ (n :: nil)) :: nil
                ⇒ if (StringDec.eq_dec gcfg c) then Some n else None
| Dpre, (term_Ctor c Czero _ nil) :: nil
                ⇒ if (StringDec.eq_dec gcfg c) then Some (fzero c) else None
| Dfst, (term_Ctor c Cpair _ (x :: y :: nil)) :: nil
                ⇒ Some x
| Dsnd, (term_Ctor c Cpair _ (x :: y :: nil)) :: nil
                ⇒ Some y
| Ddeca, (term_Ctor c1 Cenca _ (m :: (term_Ctor c2 Cpk _ (kp1 :: nil)) :: nil)) ::
         (term_Ctor c3 Csk _ (kp2 :: nil)) :: nil
                ⇒ if bool_of (TermDec.eq_dec kp1 kp2)
                     && bool_of (StringDec.eq_dec gcfg c1)
                     && bool_of (StringDec.eq_dec gcfg c2)
                     && bool_of (StringDec.eq_dec gcfg c3)
                  then Some m else None
| Dmsg, (term_Ctor c1 Csign _
                    (m :: (term_Ctor c2 Csigk _ (kp :: nil)) :: nil)) :: nil
                ⇒ if bool_of (StringDec.eq_dec gcfg c1)
                     && bool_of (StringDec.eq_dec gcfg c2)
                  then Some m else None
| Dver, (term_Ctor c1 Csign _ (m :: (term_Ctor c2 Csigk _ (kp1 :: nil)) :: nil)) ::
        (term_Ctor c3 Cvk _ (kp2 :: nil)) :: nil
                ⇒ if bool_of (TermDec.eq_dec kp1 kp2)
                     && bool_of (StringDec.eq_dec gcfg c1)
                     && bool_of (StringDec.eq_dec gcfg c2)
                     && bool_of (StringDec.eq_dec gcfg c3)
                  then Some m else None
| _, _ ⇒ None
end.
```

Listing 9: Default Destructors

2.5. Type System

The Extensible Spi Calculus has an expressive and flexible type system with subtyping and parametric polymorphism [CG92, Pie02]. This allows us to use a small number of "generically" typed constructors and destructors and still be able to specialize them. The parametric types can be nested, which naturally corresponds to the types of the nested terms and allows us to keep more information about the inner terms even after several destructor or constructor applications. The nested types also allow for expressing the relation between the type of a channel and the type of messages sent and received over it, or modeling the fact that an encryption and the corresponding key work on messages of the same type, using only a few type constructors [Bus08].

```
Section ParametrizedByName.
```

```
Inductive  type : Set :=
 |  type_Top : type
 |  type_Var : string → type
 |  type_Channel : cfg_name → type → type
 |  type_Nested : cfg_name → Idents → list type → type.

Inductive  variance : Set :=
 |  VCo : variance
 |  VContra : variance
 |  VIn : variance.
Definition  var_list : Set := (list variance).
Definition  var_list_option : Set := option (list variance).

Inductive  fun_type : Set :=
 |  ftype : list string → list type → type → fun_type.

End ParametrizedByName.
```

Listing 10: Expi Type

This type system has two fixed types, the top type `type_Top` and the type of channels `type_Channel`. These two types are required for the correct typing of processes. The type variable `type_Var` can only be used in functional types of parametric constructors and destructors (`fun_type`). The types of well-typed terms do not contain type variables, because the parametric functional types are instantiated in constructor and destructor applications. Instantiating a parametric type means replacing all type variables by the corresponding (fully instantiated) type annotations.

The type `type_Nested` represents a user-defined parametric type. It is parameterized by a type identifier from the set of identifiers `Idents` defined in the configuration (see Section 2.3) and a list of nested types. Additionally, types `type_Channel` and `type_Nested` are given a configuration name `cfg_name` which allows for defining subsets of related types such as TCP channels, AES keys and AES-encrypted messages. The configuration name can be used by the destructor reduction relation `g_red`. Our default destructor reduction relation `g_red_default` checks that the types of all arguments of a destructor have the same configuration name. This ensures for example that an AES-encrypted message cannot be decrypted with a DES key.

The set of nested types is defined using the configuration. The list `t_idents` defines which identifiers from the set `Idents` are meant to be used as type identifiers. An example for the set `Idents` is the set `Names` we use in the default configuration defined in Section 2.3.2. The partial function `t_varmap` defines the number of type parameters for each nested type and their *variance* (see Pierce, TAPL [Pie02], Chapter 15.2). The variance controls the sense of the subtyping relation (see Section 2.6). It is reversed for contravariant (`VContra`) parameters, runs in the same direction for covariant (`VCo`) parameters and requires the invariant (`VIn`) parameters to be the same. The boolean function `is_t_gen` defines which types are *generative*. Only generative types can be used in restriction processes (`proc_new`). This distinction is important in the implementation, since it is hard to provide an implementation that generates a fresh value for certain types.

The channel type `type_Channel` is assumed to be generative and have one invariant parameter.

The functional type `fun_type` is used to define the type of parametric constructors and destructors. The list of strings is assumed to contain the names of all type variables used in the argument types and the return type. For example, the type of constructor `fpair`: $\forall X, Y.\ (X, Y) \to Pair\langle X, Y\rangle$ is written as follows:

```
ftype ["X", "Y"] [type_Var "X", type_Var "Y"]
                 (type_Nested "" Pair [type_Var "X", type_Var "Y"])
```

We define a set of default types commonly used in cryptographic protocols as a part of our default configuration (see Section 2.3.2).

```
Definition  t_varmap_default (t : Names) : option (list variance) :=
match t with
 | TBool ⇒ Some nil
 | TInt ⇒ Some nil
 | TSymEnc ⇒ Some (VIn :: nil)
 | TSymKey ⇒ Some (VContra :: nil)
 | TPair ⇒ Some (VCo :: VCo :: nil)
 | THash ⇒ Some (VIn :: nil)
 | TPubEnc ⇒ Some (VIn :: nil)
 | TKeyPair ⇒ Some (VIn :: nil)
 | TPubKey ⇒ Some (VContra :: nil)
 | TPrivKey ⇒ Some (VContra :: nil)
 | TSigned ⇒ Some (VIn :: nil)
 | TSigKey ⇒ Some (VContra :: nil)
 | TVerKey ⇒ Some (VContra :: nil)
 | _ ⇒ None
end.

Definition  is_t_gen_default (t : type) : bool :=
match t with
 | type_Channel _ _ ⇒ true
 | type_Nested _ TInt nil ⇒ true
 | type_Nested _ TSymKey (_ :: nil) ⇒ true
 | type_Nested _ TKeyPair (_ :: nil) ⇒ true
 | _ ⇒ false
end.

Definition  TX s := @type_Var Names s.
Definition  Top := @type_Top Names.
Definition  Channel c X := @type_Channel Names c X.
Definition  Bool c := type_Nested c TBool nil.
Definition  Int c := type_Nested c TInt nil.
Definition  SymEnc c X := type_Nested c TSymEnc (X :: nil).
Definition  SymKey c X := type_Nested c TSymKey (X :: nil).
Definition  Pair c X Y := type_Nested c TPair (X :: Y :: nil).
Definition  Hash c X := type_Nested c THash (X :: nil).
Definition  PubEnc c X := type_Nested c TPubEnc (X :: nil).
Definition  KeyPair c X := type_Nested c TKeyPair (X :: nil).
Definition  PubKey c X := type_Nested c TPubKey (X :: nil).
Definition  PrivKey c X := type_Nested c TPrivKey (X :: nil).
Definition  Signed c X := type_Nested c TSigned (X :: nil).
```

```
Definition  SigKey c X := type_Nested c TSigKey (X :: nil).
Definition  VerKey c X := type_Nested c TVerKey (X :: nil).

Definition  f_type_default (c : cfg_name) (fn : Names)
                           : option (@fun_type Names) :=
match fn with
 | Cenc ⇒ Some (ftype ("X" :: nil) ((TX "X") :: (SymKey c (TX "X")) :: nil)
                                     (SymEnc c (TX "X")))
 | Ctru ⇒ Some (ftype nil nil (Bool c))
 | Cfls ⇒ Some (ftype nil nil (Bool c))
 | Czero ⇒ Some (ftype nil nil (Int c))
 | Csucc ⇒ Some (ftype nil ((Int c) :: nil) (Int c))
 | Cpair ⇒ Some (ftype ("X" :: "Y" :: nil) ((TX "X") :: (TX "Y") :: nil)
                                          (Pair c (TX "X") (TX "Y")))
 | Ch ⇒ Some (ftype ("X" :: nil) ((TX "X") :: nil) (Hash c (TX "X")))
 | Chmac ⇒ Some (ftype ("X" :: nil) ((TX "X") :: (SymKey c (TX "X")) :: nil)
                                     (Hash c (TX "X")))
 | Cenca ⇒ Some (ftype ("X" :: nil) ((TX "X") :: (PubKey c (TX "X")) :: nil)
                                     (PubEnc c (TX "X")))
 | Cpk ⇒ Some (ftype ("X" :: nil) ((KeyPair c (TX "X")) :: nil)
                                     (PubKey c (TX "X")))
 | Csk ⇒ Some (ftype ("X" :: nil) ((KeyPair c (TX "X")) :: nil)
                                     (PrivKey c (TX "X")))
 | Csign ⇒ Some (ftype ("X" :: nil) ((TX "X") :: (SigKey c (TX "X")) :: nil)
                                     (Signed c (TX "X")))
 | Csigk ⇒ Some (ftype ("X" :: nil) ((KeyPair c (TX "X")) :: nil)
                                     (SigKey c (TX "X")))
 | Cvk ⇒ Some (ftype ("X" :: nil) ((KeyPair c (TX "X")) :: nil)
                                     (VerKey c (TX "X")))
 | _ ⇒ None
end.

Definition  g_type_default (c : cfg_name) (gn : Names)
                           : option (@fun_type Names) :=
match gn with
 | Ddec ⇒ Some (ftype ("X" :: nil)
                      ((SymEnc c (TX "X")) :: (SymKey c (TX "X")) :: nil)
                      (TX "X"))
 | Deq ⇒ Some (ftype ("X" :: nil) ((TX "X") :: (TX "X") :: nil) (TX "X"))
 | Did ⇒ Some (ftype ("X" :: nil) ((TX "X") :: nil) (TX "X"))
 | Dpre ⇒ Some (ftype nil ((Int c) :: nil) (Int c))
 | Dfst ⇒ Some (ftype ("X" :: "Y" :: nil) ((Pair c (TX "X") (TX "Y")) :: nil)
                                          (TX "X"))
 | Dsnd ⇒ Some (ftype ("X" :: "Y" :: nil) ((Pair c (TX "X") (TX "Y")) :: nil)
                                          (TX "Y"))
 | Ddeca ⇒ Some (ftype ("X" :: nil)
                       ((PubEnc c (TX "X")) :: (PrivKey c (TX "X")) :: nil)
                       (TX "X"))
 | Dmsg ⇒ Some (ftype ("X" :: nil) ((Signed c (TX "X")) :: nil) (TX "X"))
 | Dver ⇒ Some (ftype ("X" :: nil)
                      ((Signed c (TX "X")) :: (VerKey c (TX "X")) :: nil)
                      (TX "X"))
```

```
  | _ ⇒ None
end.
```

Listing 11: Default Types

2.6. Typing Rules

The type system of the Extensible Spi Calculus is defined using several typing judgments. We call an Expi type well-formed (wf_type) if it is either the top type, an instantiated channel type or an instantiated nested type defined in the configuration.

```
Section ParametrizedByName.

Definition  is_t_defined (c : config) (t : Idents) :=
  In t (t_idents c) ∧
    ∃ lv, t_varmap c t = Some lv.

Variable C : config.

Inductive  wf_type : type → Prop :=
 |  wf_top :
     wf_type type_Top
 |  wf_channel : ∀ (cfg : cfg_name) (T : type),
     wf_type T →
     wf_type (type_Channel cfg T)
 |  wf_nested : ∀ (cfg : cfg_name) (n : Idents) (tl : list type),
     is_t_defined C n →
     wf_type_list tl →
     wf_type (type_Nested cfg n tl)
with  wf_type_list : list type → Prop :=
 |  wf_nil :
     wf_type_list nil
 |  wf_cons : ∀ (T : type) (tl : list type),
     wf_type T →
     wf_type_list tl →
     wf_type_list (T :: tl).

End ParametrizedByName.
```

Listing 12: Well-Formed Expi Type

A typing environment env is a list of bindings for free Expi names or variables. Typing environments that do not contain duplicate names and use only well-formed types are called well-formed (wf_gamma).

```
Section ParametrizedByName.

Definition  env : Set := list (atom × type).
```

```
Inductive  wf_gamma : env → Prop :=
 |  wf_empty :
     wf_gamma nil
 |  wf_update_var : ∀ (E : env) (x : atom) (T : type),
     wf_gamma E →
     wf_type T →
     x ∉ (dom E) →
     wf_gamma ((x, T) :: E)
 |  wf_update_nam : ∀ (E : env) (a : atom) (T : type),
     wf_gamma E →
     wf_type T →
     a ∉ (dom E) →
     wf_gamma ((a, T) :: E).

End ParametrizedByName.
```

Listing 13: Typing Environment Gamma

The Expi type system supports subtyping. The top type `type_Top` is a supertype of all other types. Subtyping for channel types `type_Channel` is invariant. The custom type constructors `type_Nested` are subtyped according to the corresponding variance annotations of their arguments. We have proved that the subtyping relation `syn_sub` is a partial order.

```
Section ParametrizedByName.

Variable C : config.

Inductive  syn_sub : type → type → Prop :=
 |  syn_sub_refl : ∀ (T : type),
     wf_type T →
     syn_sub T T
 |  syn_sub_top : ∀ (T : type),
     wf_type T →
     syn_sub T type_Top
 |  syn_sub_channel : ∀ (cfg : cfg_name) (T U : type),
     wf_type T →
     wf_type U →
     syn_sub T U →
     syn_sub U T →
     syn_sub (type_Channel cfg T) (type_Channel cfg U)
 |  syn_sub_nested : ∀ (cfg : cfg_name) (n : Idents) (tl ul : list type),
     wf_type_list tl →
     wf_type_list ul →
     syn_sub_list (t_varmap C n) tl ul →
     syn_sub (type_Nested cfg n tl) (type_Nested cfg n ul)
with  syn_sub_list : var_list_option → list type → list type → Prop :=
 |  syn_sub_list_nil :
     syn_sub_list (Some nil) nil nil
 |  syn_sub_list_co : ∀ (vl : var_list) (T : type) (tl : list type) (U : type)
                       (ul : list type),
     wf_type T →
```

```
    wf_type U →
    syn_sub T U →
    syn_sub_list (Some vl) tl ul →
    syn_sub_list (Some (VCo :: vl)) (T :: tl) (U :: ul)
| syn_sub_list_contra : ∀ (vl : var_list) (T : type) (tl : list type) (U : type)
                          (ul : list type),
    wf_type T →
    wf_type U →
    syn_sub U T →
    syn_sub_list (Some vl) tl ul →
    syn_sub_list (Some (VContra :: vl)) (T :: tl) (U :: ul)
| syn_sub_list_in : ∀ (vl : var_list) (T : type) (tl : list type) (U : type)
                      (ul : list type),
    wf_type T →
    wf_type U →
    syn_sub T U →
    syn_sub U T →
    syn_sub_list (Some vl) tl ul →
    syn_sub_list (Some (VIn :: vl)) (T :: tl) (U :: ul).

End ParametrizedByName.
```

Listing 14: Subtyping Relation

The type of terms is defined by the term typing relation `term_type`. The definition of `term_type` uses a predicate `binds` from the metatheory library [ACP+08] that checks if some Expi name or variable is bound to the given type in the typing environment. Another helper function used by `term_type` is `instantiate`. It instantiates the functional type of a constructor or destructor with the given type annotations by substituting all type variables present in the functional type (defined in the configuration) by the corresponding types from the type annotation.

```
Section ParametrizedByName.

Fixpoint  instantiate (annos : list type) (oft : option fun_type) {struct oft}
                    : option (list type × type) :=
match oft with
 | None ⇒ None
 | Some (ftype sl Ts T)
       ⇒ if NatDec.eq_dec (List.length sl) (List.length annos) then
              Some (subst_type_var_in_type_list (combine sl annos) Ts,
                    subst_type_var_in_type (combine sl annos) T)
          else
              None
end.
Variable C : config.

Inductive  term_type : env → term → type → Prop :=
| term_type_env_var : ∀ (E : env) (x : atom) (T : type),
    wf_gamma E →
    wf_type T →
    binds x T E →
```

```
    term_type E (term_Var_f x) T
| term_type_env_nam : ∀ (E : env) (a : atom) (T : type),
    wf_gamma E →
    wf_type T →
    binds a T E →
    term_type E (term_Nam (Nam_f a)) T
| term_type_sub : ∀ (E : env) (t : term) (U T : type),
    wf_type U →
    term_type E t T →
    syn_sub T U →
    term_type E t U
| term_type_ctor : ∀ (E : env) (cfg : cfg_name) (n : Idents) (ul : list type)
                     (xl : list term) (T : type) (tl : list type),
    wf_type_list ul →
    wf_type_list tl →
    instantiate ul (f_type C cfg n) = Some (tl, T) →
    term_type_list E xl tl →
    term_type E (term_Ctor cfg n ul xl) T
with  term_type_list : env → list term → list type → Prop :=
| term_type_list_nil : ∀ (E : env),
    wf_gamma E →
    term_type_list E nil nil
| term_type_list_cons : ∀ (E : env) (t : term) (xl : list term) (T : type)
                          (tl : list type),
    wf_gamma E →
    wf_type T →
    term_type E t T →
    term_type_list E xl tl →
    term_type_list E (t :: xl) (T :: tl).

End ParametrizedByName.
```

Listing 15: Term Typing

The processes cannot directly have a type, instead we define a process typing relation `proc_type` to check that all used terms are well-typed. This relation additionally requires the channel terms to have a channel type and the type used in restriction process `proc_new` to be generative (using the predicate `is_t_gen` from the configuration).

```
Section ParametrizedByName.

Variable C : config.

Inductive  proc_type : env → proc → Prop :=
| proc_type_out : ∀ (E : env) (t u : term) (P : proc) (T : type) (cfg : cfg_name),
    proc_type E P →
    term_type E u T →
    term_type E t (type_Channel cfg T) →
    proc_type E (proc_out t u P)
| proc_type_in : ∀ (L : vars) (E : env) (t : term) (P : proc) (cfg : cfg_name)
                   (T : type),
```

```
    term_type E t (type_Channel cfg T) →
    (∀ x, x ∉ L →
       proc_type ((x, T) :: E) (open_proc_wrt_term P (term_Var_f x))) →
    proc_type E (proc_in t P)
| proc_type_bangin : ∀ (L : vars) (E : env) (t : term) (P : proc) (cfg : cfg_name)
                       (T : type),
    term_type E t (type_Channel cfg T) →
    (∀ x, x ∉ L →
       proc_type ((x, T) :: E) (open_proc_wrt_term P (term_Var_f x))) →
    proc_type E (proc_bangin t P)
| proc_type_null : ∀ (E : env),
    wf_gamma E →
    proc_type E proc_null
| proc_type_new : ∀ (L : vars) (E : env) (T : type) (P : proc),
    is_t_gen C T = true →
    (∀ a, a ∉ L →
       proc_type ((a, T) :: E) (open_proc_wrt_nam P (Nam_f a))) →
    proc_type E (proc_new T P)
| proc_type_fork : ∀ (E : env) (P Q : proc),
    proc_type E P →
    proc_type E Q →
    proc_type E (proc_fork P Q)
| proc_type_let : ∀ (L : vars) (E : env) (cfg : cfg_name) (n : Idents)
                    (ul : list type) (xl : list term) (P Q : proc)
                    (tl : list type) (T : type),
    instantiate ul (g_type C cfg n) = Some (tl, T) →
    term_type_list E xl tl →
    (∀ x, x ∉ L →
       proc_type ((x, T) :: E) (open_proc_wrt_term P (term_Var_f x))) →
    proc_type E Q →
    proc_type E (proc_let (dtor_Dtor cfg n ul xl) P Q).

End ParametrizedByName.
```

Listing 16: Process Typing

2.7. A More Realistic Example

In this example we use the types, constructors, destructors and notations from the default configuration defined in Section 2.3.2 and Section 2.4.

```
Definition Tn : type := Int "128bit".
Definition Tenc : type := SymEnc "AES" (Tn).
Definition Tkey : type := SymKey "AES" (Tn).
Definition Tchan : type := Channel "TLS" (Tenc).

Definition n : atom := proj1_sig (atom_fresh {}).
Definition k : atom := proj1_sig (atom_fresh {{n}}).
Definition e : atom := proj1_sig (atom_fresh ({{n}} ∪ {{k}})).
Definition m : atom := proj1_sig (atom_fresh ({{n}} ∪ {{k}} ∪ {{e}})).
```

```
Definition c : atom := proj1_sig (atom_fresh ({{n}} ∪ {{k}} ∪ {{e}} ∪ {{m}})).

Definition vn : @term X := Vf n.
Definition vk : @term X := Vf k.
Definition ve : @term X := Vf e.
Definition vc : @term X := Vf c.

Definition process_a :=
  :new n ::: Tn;;  (* Generate a nonce n *)
  :out(vc, fenc "AES" Tn vn vk);;  (* Send enc(n, k) over c *)
  :0.

Definition process_b :=
  :in(vc, e);;  (* Receive e over c *)
  :let m ::= (gdec "AES" Tn ve vk) :in (  (* Decrypt e with k *)
    :0
  ) :else (
    :0
  ).

Definition process_main :=
  :new c ::: Tchan ;;  (* Generate channel c *)
  :new k ::: Tkey ;;  (* Generate a shared key k *)
  process_a :| process_b.
```

Listing 17: Typed Example

The main process starts by generating a fresh TLS channel `c` for sending encrypted nonces and a fresh shared AES key `k` for encrypting nonces. Then it starts the two participant processes `process_a` and `process_b` in parallel.

Participant A generates a fresh nonce `n` and sends it AES-encrypted with the key `k` over the channel `c`. Participant B receives the encryption `e` on channel `c` and decrypts it with the key `k`.

Note that the *enc* constructor and the *dec* destructor are given a type annotation `Tn` that is used to instantiate the corresponding type variable of their functional types.

2.8. Destructor Consistency Proof

The flexible nature of the Extensible Spi Calculus makes it impossible to prove the type system sound for all possible configurations. The destructor reduction rules in the configuration can be defined in a way that conflicts with the functional type of the destructor or some of constructors used in the reduction rules. For example, it is possible to give an identity destructor that simply returns its only argument the type $(Int) \rightarrow Bool$. Such destructor would be inconsistent, because we cannot give some term two different types that are not even subtypes of each other.

We say that a configuration is consistent if it fulfills the consistency assumptions defined in Section 2.3.1 and additionally all its destructors are consistent. A destructor is consistent, if

all its reduction rules are consistent. A reduction rule is consistent if the resulting term of a successful destructor application using this reduction rule can be typed to the return type of the destructor, assuming that the arguments have the corresponding types.

We have proved the consistency of the default configuration defined in Section 2.3.2.

```
Definition  is_dtor_consistent (g : Idents) :=
  ∀ c args ret tvars targs tret gamma,
    In g (g_idents C) →
    g_red C (dtor_Dtor c g tvars args) = Some ret →
    instantiate tvars (g_type C c g) = Some (targs, tret) →
    term_type_list gamma args targs →
      term_type gamma ret tret.

Theorem  dtor_consistency : ∀ g,
  is_dtor_consistent Names config_default g.
```

Listing 18: Destructor Consistency Theorem

The proof of `dtor_consistency` is by case analysis on the destructor identifier. The two interesting cases are the ones for the destructors `gpre` and `gdec`. Here we need to prove inversion lemmas for typing the arguments of constructors `fsucc` and `fenc` respectively. We prove these lemmas by induction on the subtyping relation and distinguishing the cases where the constructor application has its type by either subsumption rule `term_type_sub` or the constructor typing rule `term_type_ctor`. Proving `gpre` consistent requires considering two reduction rules for the `fzero` and `fsucc` cases and showing a helper lemma stating that any subtype of `Int` is also an `Int`. In the case of `gdec` destructor we do an induction on the subtyping relation of the arguments and prove some additional helper lemmas stating the preservation of the configuration for arguments of `fenc`.

The destructor consistency proof is a crucial step in the subject-reduction proof from author's bachelor's thesis.

2.9. An Attempt to Automate Proving Destructor Consistency

Destructor consistency must be proven for each new configuration. This process is quite time-consuming and can significantly complicate even a small extension of a configuration. It is very tempting to omit this step when using the expi2java tool in practice, which might result in incorrect or ill-typed implementations of protocols generated from broken models.

It would be much better if we could automate the destructor consistency check while preserving the flexibility of the calculus. This goal can be achieved in different ways, for example we could try to define an algorithm (tactic) to generate Coq proofs for any destructor definition. We have investigated another, simpler approach – to use an automatic theorem prover to reject inconsistent destructors.

The main idea was to define a simple semantics of Expi types as sets of terms and interpret subtyping as set inclusion, represent this set-theoretic semantics in higher-order logic (HOL), and then use one (or more) automatic theorem provers for HOL to prove meta-properties of this interpretation. We could not use first-order logic, because it is not expressive enough to encode parametric polymorphism and the destructor consistency meta-property.

We wanted to use this idea to automatically check that destructors are consistent by running an external HOL prover from expi2java before the type checking phase. The tool was extended to generate an input file for a HOL prover (after some testing we decided to use LEO-2 [BTPF08]), run it and interpret the result.

Unfortunately, this attempt has failed. First of all, it turned out that the complexity of the problem we tried to solve was far too high for the currently available automatic HOL provers. In most cases, the prover would not terminate in a reasonable time frame or fail to neither prove nor disprove the statement.

The more important reason for this failure was that our naïve technique was unsound. Our types can be inhabited by names, so the semantics of a type is not just a set of terms (as we naïvely thought), but a binary relation between typing environments and terms. The semantics of typing environments is however a map from names to semantic types. So intuitively this forces us to define semantic types using the following recursive equation:

$$SemType = (Name \to SemType) \to Term \to bool$$

Using a simple cardinality argument one can show that this equation does not have any set-theoretic solution. Domain theoretic or step-indexed models [AM01, Ahm04, HS09] could be a solution for this recursive equation, but encoding such things in HOL in a way which is amenable to automation would be much more challenging.

2.10. Formalization Notes

We use Ott [SNO+10] and LNgen [AW10] to generate the Coq definitions of our formalization of the Extensible Spi Calculus. Ott is a tool for writing definitions of programming languages and calculi. It has a nice, concise syntax and natively supports locally-nameless representation of variables and names. LNgen is another tool that generates "infrastructure lemmas" for the Coq definitions generated from Ott specification. This locally-nameless infrastructure consists of a so called "metatheory library" [ACP+08] and a large collection of lemmas generated specifically for the formalization produced by Ott. The lemmas and Coq tactics provided by LNgen are very helpful for proving various properties of the generated calculus.

Unfortunately, neither Ott nor LNgen are perfect. We had to additionally patch their output to produce a correct formalization. First of all, none of them allow to parameterize the generated formalization with some variables. We had to enclose the generated code into a section with several Coq variables to model an extensible language that can only be fully defined using an additional configuration (see Section 2.3). Furthermore, Ott seems to have problems with languages that have more than one locally-nameless metavariable (i.e., Expi names and variables in our case). We have found a workaround that does not give an error,

but some of the generated definitions were still wrong, we had to fix them manually. The biggest problem with LNgen was the lack of support for lists. We use lists of terms and lists of types in many definitions, and the only way to make Ott produce correct definitions in these cases is to use the native Ott lists. However, we had to replace those lists by Coq lists for LNgen, which leads to missing lemmas for that cases. We had to patch the generated infrastructure lemmas to make them pass the Coq type-checker.

3. Formalizing Variant Parametric Jinja (VPJ)

The target language is a substantial subset of Java with an alternative type system that supports variant parametric types. This language is based on the subset of Java called "Jinja with Threads" developed by A. Lochbihler [Loc08] and the type system is based on Variant Parametric Types by A. Igarashi and M. Viroli [IV06]. We will refer to this Java subset as *Variant Parametric Jinja (VPJ)*. In this chapter we will define the syntax, type system, and the operational semantics of VPJ.

3.1. Jinja with Threads

Jinja with threads is a subset of Java formalized in the proof assistant Isabelle/HOL [PNW11]. It is an extension of Jinja, a single-threaded subset of Java developed by G. Klein and T. Nipkow [KN06]. To our best knowledge, Jinja with Threads is the most complete and thorough formalization of Java that supports concurrency, which made it the first candidate for a starting point for our own formalization. In this section, we will introduce the features of Jinja with Threads and briefly explain its syntax and semantics with an emphasis on the modifications we have made. Please refer to the corresponding papers for more details [Loc08, KN06, IV06].

The subset of Java suitable to model Extensible Spi Calculus has to be very expressive. We need concurrency with shared memory and synchronization to model channels and message passing, a class hierarchy with inheritance and casting to model data types, a type system that at least supports Java generics and optionally, exceptions to simplify modeling destructor applications. Jinja with Threads supports all these features except for generics, that was the main reason for choosing Jinja with Threads over other formalizations of Java.

Both Jinja and Jinja with Threads are formalized in Isabelle/HOL. The aim of these projects is to make a solid formalization of Java together with the corresponding Java Virtual Machine (JVM) and a compiler from Java to byte code. The authors prove type safety of Java and JVM and correctness of the compiler. We have only used the formalization of Java and disregarded any further steps towards the byte code, the JVM semantics, and all proofs. In order to be able to use Jinja with our formalization of the Extensible Spi Calculus, we have manually translated the formalization of Jinja with Threads from Isabelle/HOL to Coq.

We have slightly modified Jinja with Threads to better suit our needs. First of all, we have extended the values and types with support for a new base type, `String`. We did this to simplify storing various names (constructors, configuration name etc.). We have also removed the support for arrays that was introduced in Jinja with Threads to simplify the semantics. The biggest change was however the introduction of variant parametric types needed to model

the parametric polymorphism of the Expi type system. We will explain variant parametric types in more details in Section 3.3.

The switch to Coq has also had an impact on the way the language is formalized. Isabelle/HOL heavily relies on implicit behavior to complete the definitions and so gives the user a lot more freedom. For example, Isabelle makes it possible to use a partial function (e.g., of type $nat \to option\ nat$) as a total one (i.e., of type $nat \to nat$), it automatically uses a special "undefined" value where needed. This is not possible in Coq, here one has to deal with all the cases explicitly. Another difference is that Isabelle/HOL uses classical logic, so laws such as excluded middle and functional extensionality are built deep into the framework and used pervasively. On the other hand Coq's logic is constructive by default (based on the Calculus of Inductive Constructions [CH88, PPM90, CPM90]), and while one can import the classical axioms in Coq, working with such axioms is not always straightforward.

Because of these differences we had to define parts of the formalization in a different manner that is more suitable for Coq. In particular, we replaced lists used to store declarations of classes, methods, fields etc. by functional maps. We avoided comparing functions, because we would need to use functional extensionality and defined relations that express the intended result instead. In two cases, however, we could not find an easy way to stay constructive and used classical axioms (excluded middle and constructive indefinite description): in the definition of the multi-threaded reduction relation and in the heap allocation function `dec_alloc`.

3.2. Variant Parametric Featherweight Java

Jinja with Threads lacks one feature that is important to us – an expressive type system with support for parametric types. The parametric polymorphism on the Java side is needed to simplify the encoding of the polymorphic type constructors in Extensible Spi Calculus. We have chosen to use variant parametric types [IV06] as the basis for our formalization. This very expressive type system is used in Variant Parametric Featherweight Java, an extension of Featherweight GJ [IPW01], itself an extension of Featherweight Java [IPW01]. The variant parametric types of Igarashi and Viroli [IV06] are intimately related to Java wildcards and generics [THE+04], as implemented in Java™ version 5.0[1].

Variant Parametric Featherweight Java is a very small functional subset of Java that is designed to study type systems rather than being used to write programs. This language is very abstract and has almost no features we need, i.e., no concurrency, no exceptions, not even state. The only interesting feature is the type system, it is even more expressive than we need, because the variance of the type parameters is not fixed by the class declaration, but can be specified on class instantiation. Variant Parametric Types allow to parameterize classes and methods with type variables with bounds and specify variance of those type variables when such classes are instantiated. The authors prove soundness of this type system. The formalization of Variant Parametric Featherweight Java is not machine-checked, we have translated it to Coq and adapted it for use with our model of Jinja.

[1] http://download.oracle.com/javase/tutorial/extra/generics/wildcards.html

3.3. Formalization Overview

The formalization of VPJ is larger and more complex than the one of Expi. In the next sections we will explain the most important parts of it and show excerpts of the more complicated definitions. Please refer to Appendix A for the complete Coq code listings of the corresponding definitions.

The diagram in Figure 3.1 shows a simplified dependency graph of the formalization, where each node represents a Coq file.

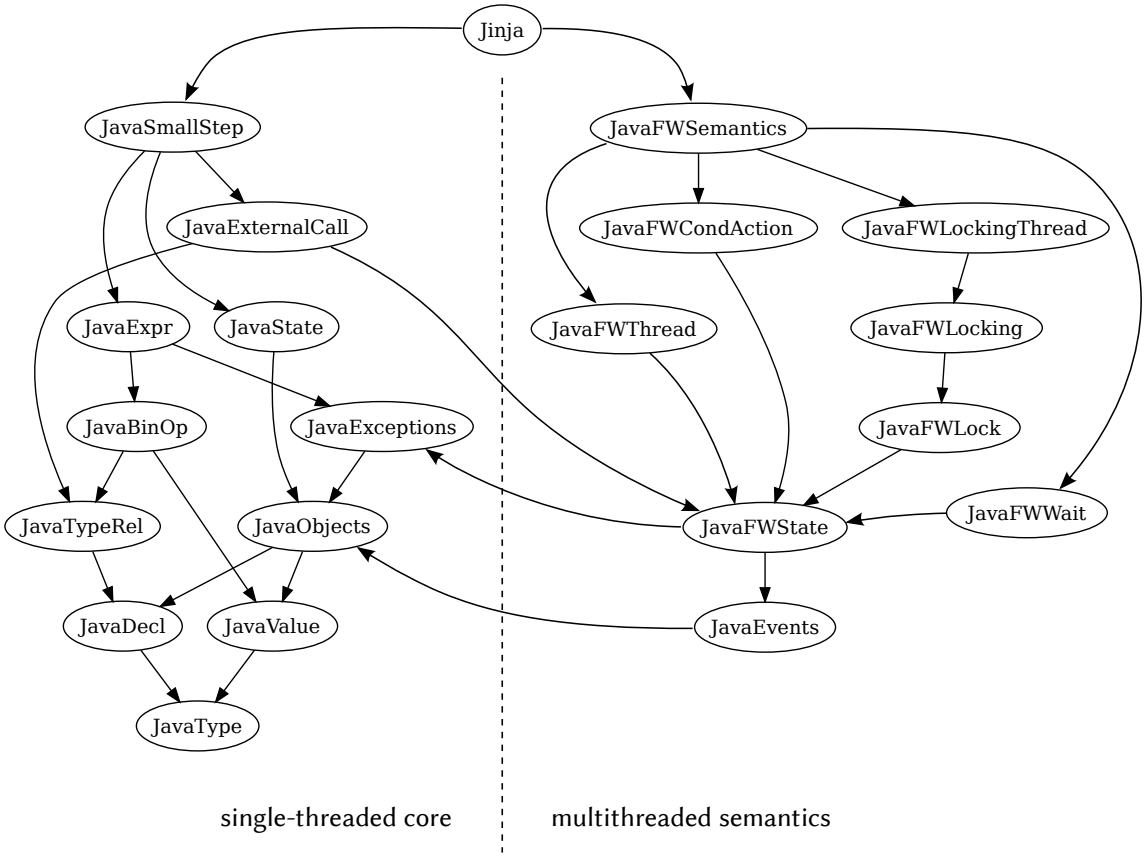


Figure 3.1.: VPJ Formalization Overview

The formalization of VPJ can be split into two parts, the *single-threaded core* and the *multi-threaded semantics*. The single-threaded core defines the syntax and single-threaded small-step semantics of VPJ. This is the part we have extended with the variant parametric types. The multi-threaded semantics is exactly the same as in Jinja with Threads, we have only made changes that had to be done to adapt the formalization for Coq. We will focus on the single-threaded part, emphasizing our modifications and give only a brief overview over the multi-threaded part.

3.4. Syntax of VPJ

The names of classes, methods, local variables, fields and type variables are represented using Coq strings. To improve readability, we use Coq definitions for different names and also the names of various standard classes and method names such as `Object` and `this`.

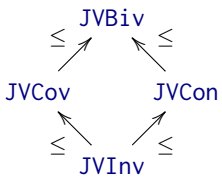
```
Definition  cname := string.   (* Class names *)
Definition  mname := string.   (* Method names *)
Definition  vname := string.   (* Fields and local variables *)
Definition  tvname := string.   (* Type variable names *)

(* Names of standard classes *)
Definition  Object := "Object" : cname.
Definition  Thread := "Thread" : cname.
Definition  Throwable := "Throwable" : cname.

Definition  this := "this" : vname.

(* Names of thread control methods *)
Definition  run := "run" : mname.
Definition  start := "start" : mname.
Definition  wait := "wait" : mname.
Definition  notify := "notify" : mname.
Definition  notifyAll := "notifyAll" : mname.
Definition  join := "join" : mname.
```

Listing 19: VPJ Type Definitions

3.4.1. Types

Variant parametric types use variance annotations, similar to the `variance` from Extensible Spi Calculus. The difference to Expi type system is that the variance in VPJ can also be *bivariant*. The subtyping relation for bivariant parameters is satisfied if one of the parameters is a subtype of the other (i.e., bivariance is the disjunction of co- and contravariance, while invariance is the conjunction of co- and contravariance). This implies that any two instantiations of a bivariant type constructor are subtypes of each-other, no matter in what relations are the arguments of the type constructor.

We define a $\leq$ relation `varsmaller` on `jvariance` (illustrated by the diagram on the right) and a corresponding least upper bound function `varlub`. The $\leq$ relation is reflexive, antisymmetric and transitive. It is used together with `varlub` to define the subtyping relation for VPJ types, defined in Section 3.5.




```
Inductive  jvariance : Type :=
 |  JVInv  (* invariant:  o *)
 |  JVCov  (* covariant:  + *)
 |  JVCon  (* contravariant:  - *)
```

```
 | JVBiv.   (* bivariant:  * *)

Inductive  varsmaller : jvariance → jvariance → Prop :=
 | vs_refl : ∀ v, varsmaller v v
 | vs_inv_cov : varsmaller JVInv JVCov
 | vs_inv_con : varsmaller JVInv JVCon
 | vs_cov_biv : varsmaller JVCov JVBiv
 | vs_con_biv : varsmaller JVCon JVBiv
 | vs_inv_biv : varsmaller JVInv JVBiv.

Definition  varlub (v w : jvariance) : jvariance :=
match v, w with
 | JVInv, _ ⇒ w
 | JVCov, JVInv ⇒ JVCov
 | JVCov, JVCov ⇒ JVCov
 | JVCov, _ ⇒ JVBiv
 | JVCon, JVInv ⇒ JVCon
 | JVCon, JVCon ⇒ JVCon
 | JVCon, _ ⇒ JVBiv
 | JVBiv, _ ⇒ JVBiv
end.
```

Listing 20: Type Variance in VPJ

The definition of a VPJ type is split into 3 inductive definitions. This allows to distinguish between different subsets of VPJ types in typing judgments. The type of a class is modeled by `jclass_type`, parameterized with the corresponding class name and a list of type parameters together with their variances. A type parameter `jparam_type` can either be a type variable `TV` (parameterized with its name) or a class type `TC`. Finally, `jtype` represents the VPJ type, which can either be one of the primitive types (`Void`, `Boolean`, `Integer` or `String`), the type of a null value `NullType` or a parameter type `RefType`. The types `NullType` and `RefType` are also called *reference types* in Java.

We define several notations to simplify defining types in common cases, such as the `jtype` of a class or a type variable. Additionally, we use `InvTClass` and `InvClass` to define class types, invariant in all parameters. This case is often used in class declarations and other cases, where variance annotations are not used.

```
Inductive  jclass_type : Type :=
 | TClass : cname → list (jvariance × jparam_type) → jclass_type
with  jparam_type : Type :=
 | TV : tvname → jparam_type
 | TC : jclass_type → jparam_type.

Inductive  jtype : Type :=
 | Void
 | Boolean
 | Integer
 | String
 | NullType
```

```
| RefType : jparam_type → jtype.

(* Notations *)
Definition  Class cn params : jtype := RefType (TC (TClass cn params)).
Definition  TVar tvn : jtype := RefType (TV tvn).

Definition  CObject : jclass_type := TClass Object nil.
Definition  CThread : jclass_type := TClass Thread nil.
Definition  CThrowable : jclass_type := TClass Throwable nil.

Definition  TObject := RefType (TC CObject).
Definition  TThread := RefType (TC CThread).
Definition  TThrowable := RefType (TC CThrowable).

Definition  InvTClass (cn : cname) (ps : list jparam_type) : jclass_type :=
  TClass cn (map (fun x ⇒ (JVInv, x)) ps).

Definition  InvClass cn ps : jtype :=
  RefType (TC (InvTClass cn ps)).
```

Listing 21: VPJ Types

3.4.2. Declarations

Just like Java, Variant Parametric Jinja allows to declare custom classes with methods and fields. We use functional maps to let the user define finite sets of named declarations. Functional maps are partial functions that take a name (of type string) as an argument and return `Some` *value* in the case that the given name is one of the defined ones and `None` otherwise. Usually, the `option` type is used as the return type to define partial functions in Coq. We use a bit more complicated construct instead, an *error monad* `Maybe`. It uses the `option` type internally, but also defines some useful functions and lemmas that simplify more complex use cases. We will briefly explain error monad functions when needed, see Appendix A.2 for more details.

The field declaration `fdecl` is a functional map that maps field names to their types. Similarly, the method declaration `mdecl` maps method names to the corresponding method definitions. A method definition `mdef` is a tuple consisting of a list of method type parameter names with their upper bounds, a list of parameter names and their types, the return type and the method body expression. Finally, a VPJ program `prog` is a partial function that maps class names to the corresponding class definitions `class`. A class definition is a tuple of a list of class type parameters with their upper bounds, the supertype of this class, a list of field names, the field declarations and the method declarations.

Please note how the different VPJ type definitions are used to narrow down the set of types allowed in different cases. For example, fields and method parameters can have any type represented by `jtype`, but the superclass and the upper bounds of the class type parameters can only have the class type `jclass_type`. The upper bounds of the method type parameters have the type `jparam_type`, because both class type and type variables (the ones defined in the class) can be used in this case.

```
Section  Java.

(* Instantiated with expression on use (needed to break circular dependency) *)
Variable  E : Type.

Definition  fdecl := vname → Maybe jtype.
Definition  mdef := (list (tvname × jparam_type))
                    × (list (vname × jtype))
                    × jtype
                    × E.
Definition  mdecl := mname → Maybe mdef.
Definition  class := (list (tvname × jclass_type))
                    × jclass_type
                    × (list vname × fdecl)
                    × mdecl.
Definition  prog := cname → Maybe class.

End Java.
```

Listing 22: VPJ Declarations

3.4.3. Values

The definition of Jinja values is quite straightforward. `Unit` stands for a dummy value of type `Void`, `Null` is the analog of Java's *null* value, `Bool`, `Intg` and `Str` map values of built-in Coq types `bool`, `Z` (signed integer) and `string` to the corresponding VPJ types[2]. Finally, `Addr` stands for a value of a memory location `addr`, represented by a natural number. The memory model of VPJ is presented in more details in Section 3.4.4, for now we just need to know that each newly allocated object is assigned a fresh memory location. This freshness property is used to distinguish threads (instances of class `Thread`) by their `thread_id`.

We also define the function `default_val` that returns the default value for a given VPJ type. This function is used to initialize fields and local variables according to the Java specification [LY99].

```
Definition  addr := nat.
Definition  thread_id := addr.

Inductive  val : Type :=
 |  Unit
 |  Null
 |  Bool : bool → val
 |  Intg : Z → val
 |  Str : string → val
 |  Addr : addr → val.

Fixpoint  default_val (T : jtype) : val :=
```

[2] Note that unlike Java integers, the Coq's signed integers are unbounded, we use them just for the sake of simplicity, our transformation never actually uses integer expressions.

```
match T with
 | Void ⇒ Unit
 | Boolean ⇒ Bool false
 | Integer ⇒ Intg 0
 | String ⇒ Str ""
 | NullType ⇒ Null
 | RefType _ ⇒ Null
end.
```

Listing 23: VPJ Values

3.4.4. Memory Model

Variant Parametric Jinja uses a realistic memory model with a shared heap and thread-local stacks. We model both heap and stack using functional maps.

The heap is a partial function that maps memory locations `addr` to heap objects `heapobj`. A heap object contains the class type of the corresponding instance and a functional map of its field values `fields`. This semantics differs quite a bit from the semantics used in Java. The difference between them is that our formalization stores the exact parametric type of each object, while in Java, the information about the type parameters is lost.[3] We have decided to store the parametric types to simplify the formalization of variant parametric types and avoid the problems arising from Java-like semantics, such as the need for run-time type checks.

The stack `locals` is a simple mapping from variable names to value. Finally, the state of a program `Jstate` is a tuple consisting of heap and stack.

The mapping `fields` maps from the pair `vname × cname` to values `val`. We additionally need the name of the class where the field was defined to model *field shadowing*. A superclass field can be "shadowed" in a subclass if it declares a field with the same name. According to the Java specification, these fields must be distinct and independently accessible. In Java, the type of the object used in the field access expression determines which of them is accessed, Jinja uses a simpler semantics (also used in Java bytecode) and require providing the class name explicitly.

We prove that the allocation on the heap (i.e., checking if the set of free memory locations is inhabited) is decidable and use the resulting lemma `dec_alloc` to define the memory allocation function `new_Addr`. This definition is quite abstract and does not explain how exactly the new memory location is found, but it is sufficient in our case, because we are just interested in an allocation function that *can* either succeed or fail so that we have to cover both cases in the proofs.

```
Definition  fields := vname × cname → Maybe val.

Inductive  heapobj : Type :=
 |  Obj : jclass_type → fields → heapobj.
```

[3] In the Java community, this process is usually called "type erasure", although not all type information is actually erased.

```
Definition  heap := addr → Maybe heapobj.

Definition  dec_alloc : ∀ h : heap, {a : addr | h a = None} + {∀ a, h a ≠ None}.

Definition  new_Addr (h : heap) : Maybe addr :=
match dec_alloc h with
 | inleft H ⇒ Some (proj1_sig H)
 | inright _ ⇒ None
end.

Definition  locals := vname → Maybe val.

Definition  Jstate := heap × locals.
```

Listing 24: Heap and Stack

3.4.5. System Exceptions

VPJ uses several *system exceptions* to handle various error conditions that are found during the reduction (see Section 3.6 for more details). The system exceptions are defined by the inductive declaration `SysException`. The names of these exceptions are the same as in Java and do not need any further explanations. In addition to the inductively defined set of exceptions, we reserve the corresponding class names and VPJ types. All system exceptions are direct subclasses of the `Throwable` class.

```
Inductive  SysException : Type :=
 |  eNullPointer
 |  eClassCast
 |  eOutOfMemory
 |  eIllegalMonitorState
 |  eIllegalThreadState.

Definition  NullPointer : cname := "NullPointer".
Definition  ClassCast : cname := "ClassCast".
Definition  OutOfMemory : cname := "OutOfMemory".
Definition  IllegalMonitorState : cname := "IllegalMonitorState".
Definition  IllegalThreadState : cname := "IllegalThreadState".
```

Listing 25: System Exceptions

We assume that all system exceptions are *preallocated* on the heap before a VPJ program is executed. This allows simpler exception handling since we can distinguish system exceptions from custom user exceptions just by comparing their address. The corresponding class and address of a `SysException` can be obtained using the helper functions `class_of_sys_exc` and `addr_of_sys_exc`.

```
Definition  CNullPointer := TClass NullPointer nil.
Definition  CClassCast := TClass ClassCast nil.
```

```
Definition  COutOfMemory := TClass OutOfMemory nil.
Definition  CIllegalMonitorState := TClass IllegalMonitorState nil.
Definition  CIllegalThreadState := TClass IllegalThreadState nil.

Fixpoint  class_of_sys_exc (e : SysException) {struct e} : jclass_type :=
match e with
 | eNullPointer ⇒ CNullPointer
 | eClassCast ⇒ CClassCast
 | eOutOfMemory ⇒ COutOfMemory
 | eIllegalMonitorState ⇒ CIllegalMonitorState
 | eIllegalThreadState ⇒ CIllegalThreadState
end.

Fixpoint  addr_of_sys_exc (e : SysException) {struct e} : addr :=
match e with
 | eNullPointer ⇒ 0
 | eClassCast ⇒ 1
 | eOutOfMemory ⇒ 2
 | eIllegalMonitorState ⇒ 3
 | eIllegalThreadState ⇒ 4
end.

Definition  is_preallocated (h : heap) : Prop :=
 ∀ e, ∃ fs, h (addr_of_sys_exc e) = Some (Obj (class_of_sys_exc e) fs).
```

Listing 26: Exception Allocation

3.4.6. Binary Operations

Variant Parametric Jinja supports various binary operations. Two values can be tested for equality using the operators `Eq` and `NotEq`. We use pointer equality to compare objects, because structural equality is much harder to implement and is not always needed. Integers can also be compared to each other with `LessThan`, `LessOrEqual`, `GreaterThan` and `GreaterOrEqual`. Some simple integer arithmetics are also supported with `Add`, `Subtract` and `Mult`. Finally, boolean logic is also supported with `BoolAnd`, `BoolOr` and `BoolXor`. The typing rules for these operations are defined by the relation `WT_binop` in Appendix A.4 and their semantics is defined by the function `binop` in Appendix A.5.

```
Inductive  bop : Type :=
 |  Eq
 |  NotEq
 |  LessThan
 |  LessOrEqual
 |  GreaterThan
 |  GreaterOrEqual
 |  Add
 |  Subtract
 |  Mult
 |  BoolAnd
 |  BoolOr
```

```
| BoolXor.

Notation  "a == c" := (BinOp a Eq c) (at level 71, no associativity) : vpj_scope.
Notation  "a != c" := (BinOp a NotEq c) (at level 71, no associativity) : vpj_scope.

Notation  "a && c" := (BinOp a BoolAnd c) (at level 40, left associativity) : vpj_scope.
Notation  "a || c" := (BinOp a BoolOr c) (at level 50, left associativity) : vpj_scope.
Notation  "a ^^ c" := (BinOp a BoolXor c) (at level 40, no associativity) : vpj_scope.
```

Listing 27: Binary Operations

In order to improve readability of VPJ code we define Coq notations for boolean operations. The syntax of these notations is the same as for the corresponding expressions in Java.

3.4.7. Expressions

VPJ expressions essentially combine the most important Java expressions and statements into one language construct. We can create new instances of objects using the `New` expression, but unlike in Java we have to provide a list of expressions used to initialize all the fields instead of calling a constructor. This list is not present in the original Jinja formalization [KN06] where all fields are initialized with their default values (see `default_val`). `Cast` represents the usual casting expression used to change the type of an expression to its super- or subtype. We remark that the cast cannot be used to convert primitive types (e.g., **int** to **float**) like in Java. `Val` simply converts a VPJ value (see `val`) into an expression. Binary operations are performed using the `BinOp` expression.

`Var` stands for local variable access and `LAss` for assignments to local variables. The fields are read using `FAcc` and written with `FAss`. Note that the field access needs to know the name of the class where the field is defined in addition to the usual object expression and field name. This is needed to allow accesses to shadowed fields. Field accesses on the Java byte code level happen in exactly the same way, the class name is not present in Java syntax because the compiler infers it.

`Call` is used to call methods, which can be parametric. The user has to provide the object expression whose method is called, the list of type annotations used to instantiate the method type parameters, the method name and the list of argument expressions. The `Block` expression is the way to declare a local variable in VPJ. Local variables can be initialized to some value and are only visible in the scope of their block, i.e., in the expression given to the `Block` expression.

`Sync` models the **synchronized** blocks from Java. The threads are synchronized on the implicit lock of the object given to the `Sync` expression and protects the subexpression. `InSync` represents an intermediate step produced by the reduction relation used in the formalization of Jinja with Threads [Loc08].

The remaining expressions are quite standard. `Seq` is a sequence of two expressions (the semicolon in Java), `Cond` is the conditional expression (similar to the ternary conditional operator "_ ? _ : _" from Java), `while` and `throw` speak for themselves and the `TryCatch`

represents the try-catch construct. Note that the **finally** block is pure syntactic sugar and is not supported by VPJ.

```
Inductive  expr : Type :=
 |  New : jclass_type → list expr → expr
 |  Cast : jtype → expr → expr
 |  Val : val → expr
 |  BinOp : expr → bop → expr → expr
 |  Var : vname → expr
 |  LAss : vname → expr → expr
 |  FAcc : expr → vname → cname → expr
 |  FAss : expr → vname → cname → expr → expr
 |  Call : expr → list jparam_type → mname → list expr → expr
 |  Block : vname → jtype → Maybe val → expr → expr
 |  Sync : unit → expr → expr → expr
 |  InSync : unit → addr → expr → expr
 |  Seq : expr → expr → expr
 |  Cond : expr → expr → expr → expr
 |  while : expr → expr → expr
 |  throw : expr → expr
 |  TryCatch : expr → jclass_type → vname → expr → expr.

(* Instantiate the variable E *)
Definition  jmdef := @mdef expr.
Definition  jmdecl := @mdecl expr.
Definition  jclass := @class expr.
Definition  jprog := @prog expr.

(* Syntactic sugar *)
Definition  junit : expr := Val Unit.
Definition  jnull : expr := Val Null.
Definition  jaddr a : expr := Val (Addr a).
Definition  jtrue : expr := Val (Bool true).
Definition  jfalse : expr := Val (Bool false).
Definition  jThrow a : expr := throw (Val (Addr a)).
Definition  jTHROW xc : expr := jThrow (addr_of_sys_exc xc).
Definition  jsync e1 e2 : expr := Sync tt e1 e2.
Definition  jinsync (a : addr) e2 : expr := InSync tt a e2.
Definition  jthis : expr := Var "this".
Definition  new (cn : cname) (cps : list jparam_type)
               (fparams params : list expr) : expr :=
  Call (New (InvTClass cn cps) fparams) nil cn (params).

Notation  "x ::= e" := (LAss x e) (at level 57, right associativity) : vpj_scope.
Notation  "e1 ; e2" := (Seq e1 e2) (at level 60, right associativity) : vpj_scope.
Notation  ":if e1 :then e2 :else e3" := (Cond e1 e2 e3)
                                        (at level 60, no associativity) : vpj_scope.
Notation  ":try :{ e1 :} :catch C :ex x :{ e2 :}" := (TryCatch e1 C x e2)
                                        (at level 60, no associativity) : vpj_scope.
```

Listing 28: VPJ Expressions

We define some syntactic sugar to simplify writing VPJ programs. It consists of repeatedly occurring expressions such as `jthis`, `jtrue` and `jnull`, a shorter way to throw expressions and write synchronized statements, and a `new` construct that mimics the behavior of the **new** statement in Java (creates an object and calls a constructor method). Additionally, we define Coq notations for some frequently used expressions, such as local variable assignment `LAss`, sequential composition `Seq`, conditional operator `Cond` and the try-catch expression `TryCatch`. We tried to mimic the syntax used in the corresponding Java constructs, as much as it was possible in Coq. Unfortunately, the abilities of Coq are quite limited in that respect.

3.5. Type System

3.5.1. Typing Judgments

The type system of Variant Parametric Jinja is a mixture between the Jinja type system and the variant parametric types [IV06]. We have both the simpler subclass relation `subclass` from Jinja that is used in cases where only non-parametric classes can be used (for example when throwing system exceptions) and a subtype relation `subtype` that is used instead of the subclass relation in cases where data is accessed (e.g., reading and writing fields).

The subclass relation is a transitive closure over the direct subclass relation `subclass1`. The direct subclass relation is defined in a straightforward way, with `Object` at the top of the class hierarchy.

```
Section Java.

(* Instantiated with expression on use (needed to break circular dependency) *)
Variable E : Type.
(* The VPJ program, all definitions below are parameterized by it *)
Variable P : @prog E.

(* Direct subclass *)
Inductive subclass1 : cname → cname → Prop :=
| subclass1_object : ∀ C tvs fds mds ps,
                       P C = Some (tvs, TClass Object ps, fds, mds) →
                       C ≠ Object →
                       subclass1 C Object
| subclass1_rec : ∀ C D ps tvs fds mds,
                    P C = Some (tvs, TClass D ps, fds, mds) →
                    C ≠ Object →
                    D ≠ Object →
                    subclass1 C D.

Definition subclass : cname → cname → Prop :=
  clos_trans_1n cname subclass1.

End Java.
```

Listing 29: Subclass Relation

The subtyping relation `subtype` is almost the same as in variant parametric types, we added only two additional rules: Rule `sub_null` states that the `NullType` is a subtype of all reference types. Rule `sub_null_str` states that the `NullType` is a subtype of the `String` type. Additionally, we require the subclass relation to be *well-founded* (i.e., that it does not have infinite descending chains) to ensure termination of a subclass check. Any well-formed VPJ program must fulfill this assumption.

The *type variable environment* `delta` (not to be confused with the *typing* environment `gamma`) maps type variables to parametric types and a variance. A type variable mapped to the pair `(JVCov, T)` is seen as a subtype of `T` (see `sub_ubound`). Similarly, a type variable mapped to the pair `(JVCon, T)` is seen as a supertype of the type `T` (see `sub_lbound`). Invariance and bivariance correspond to the conjunction and the disjunction of co- and contravariance respectively [IV06].

If two class types differ only in type parameters, the parameters are compared to each other according to their variance (see `sub_var` and `subtype_vars`). A class type can also be a subtype of another type `T` if `T` is the minimal supertype of its direct superclass with instantiated type variables (see `sub_class`). The auxiliary function `closes_to` computes the minimal supertype. The predicate `opens_to` is related to the representation of variant parametric types as existential types. These auxiliary functions are defined in Appendix A.3, please refer to the work by Igarashi et al. [IV06] for more details about this type system.

```
Definition  superclass1 (dn cn : cname) := subclass1 cn dn.

(* Type environment *)
Definition  delta := tvname → Maybe (jvariance × jparam_type).

Inductive  subtype {E : Type} : @prog E → delta → jtype → jtype → Prop :=
 |  sub_ubound : ∀ P d X T,
                 well_founded (superclass1 E P) →
                 d X = Some (JVCov, T) →
                 subtype P d (TVar X) (RefType T)
 |  sub_lbound : ∀ P d X T,
                 well_founded (superclass1 E P) →
                 d X = Some (JVCon, T) →
                 subtype P d (RefType T) (TVar X)
 |  sub_null : ∀ P d T,
                 well_founded (superclass1 E P) →
                 subtype P d NullType (RefType T)
 |  sub_null_str : ∀ P d, subtype P d NullType String
 |  sub_class : ∀ P d d' cn ctvnts D fd md Ts Us T,
                 well_founded (superclass1 E P) →
                 P cn = Some (ctvnts, D, fd, md) →
                 opens_to d (Class cn Ts) d' (InvClass cn Us) →
                 closes_to (RefType (subst_jtvars_in_jparam
                                       (map fst ctvnts) Us (TC D))) d' T →
                 subtype P d (Class cn Ts) T
 |  sub_var : ∀ P d cn vTs wUs,
                 well_founded (superclass1 E P) →
                 subtype_vars P d vTs wUs →
```

```
                subtype P d (Class cn vTs) (Class cn wUs)
 |  sub_refl : ∀ J d T, subtype J d T T
 |  sub_trans : ∀ J d T U V,
                subtype J d T U →
                subtype J d U V →
                subtype J d T V
with  subtype_vars {E : Type} : @prog E → delta →
                  list (jvariance × jparam_type) →
                  list (jvariance × jparam_type) → Prop :=
 |  sub_vars_nil : ∀ P d,
                  well_founded (superclass1 E P) →
                  subtype_vars P d nil nil
 |  sub_vars_list : ∀ P d v w T U vTs wUs,
                   well_founded (superclass1 E P) →
                   varsmaller v w →
                   (varsmaller w JVCon →
                      subtype P d (RefType U) (RefType T)) →
                   (varsmaller w JVCov →
                      subtype P d (RefType T) (RefType U)) →
                   subtype_vars P d vTs wUs →
                   subtype_vars P d ((v, T) :: vTs) ((w, U) :: wUs).
```

Listing 30: Subtyping Relation

A VPJ type is *well-formed* in the given program and a type variable environment `delta` if it is either one of the primitive types, a type variable that is contained in `delta`, the class `Object` or a custom class declared in the program.

```
Inductive  is_wf_type {E : Type} : @prog E → delta → jtype → Prop :=
 |  wf_type_void : ∀ P d, is_wf_type P d Void
 |  wf_type_bool : ∀ P d, is_wf_type P d Boolean
 |  wf_type_int : ∀ P d, is_wf_type P d Integer
 |  wf_type_str : ∀ P d, is_wf_type P d String
 |  wf_type_null : ∀ P d, is_wf_type P d NullType
 |  wf_type_object : ∀ P d x,
                     P Object = Some x →
                     is_wf_type P d (RefType (TC (TClass Object nil)))
 |  wf_type_tvar : ∀ P d X vT,
                   d X = Some vT →
                   is_wf_type P d (RefType (TV X))
 |  wf_type_class : ∀ (P : @prog E) d cn ctvnts D fns
                    (fd : fdecl) (md : mdecl) vTs,
                    P cn = Some (ctvnts, D, (fns, fd), md) →
                    is_wf_type P d (RefType (TC D)) →
                    is_wf_type_list P d vTs (map snd ctvnts) →
                    is_wf_type P d (RefType (TC (TClass cn vTs)))
with  is_wf_type_list {E : Type} : @prog E → delta →
                    list (jvariance × jparam_type) →
                    list jclass_type → Prop :=
 |  wf_type_nil : ∀ P d, is_wf_type_list P d nil nil
```

```
| wf_type_list : ∀ P d v T vTs U Us,
                  is_wf_type P d (RefType T) →
                  subtype P d (RefType T) (RefType (TC U)) →
                  is_wf_type_list P d vTs Us →
                  is_wf_type_list P d ((v, T) :: vTs) (U :: Us).
```

Listing 31: Well-Formed Types

3.5.2. Expression Typing

The *typing environment* `gamma` is a functional mapping from variable names to types. A `gamma` is well-formed if it contains only well-formed types.

```
Definition  gamma := vname → Maybe jtype.

Definition  wf_gamma (P : jprog) (d : delta) (g : gamma) : Prop :=
  ∀ x T, g x = Some T → is_wf_type P d T.
```

Listing 32: Typing Environment

The expression typing in VPJ is very similar to Jinja with small differences related to the type system. We use the subtype relation instead of the subclass relation and use `opens_to` and `closes_to` to deal with variant parametric types. We only present a short excerpt of the expression typing relation here (it has 20 rules in total), see Appendix A.4 for the complete definition.

A value `Val` is well-typed if we can infer its type without using the heap (rule `wte_val`). We cannot use the type information stored in the heap because the typing is static. As a consequence, this rule forbids direct memory accesses with the expression `Addr` in well-typed programs. A variable `Var` is well-typed if it has a well-formed type in the typing environment `gamma` (rule `wte_var`).

The `while` expression requires the condition to have the type `Boolean` and recursively checks if the body expression is well-typed too (rule `wte_while`). The try-catch block `TryCatch` declares a variable `x` for the caught exception (rule `wte_trycatch`). Therefore, the type of `x` has to be a subtype of `TThrowable`. The `wte_trycatch` rule requires both the try-expression and the catch-expression to have the same type (e.g., `Void` could be used here), but the catch-expression `e2` is typed in an updated typing environment, where the caught expression variable `x` is visible with the corresponding type.

```
Fixpoint  typeof (v : val) : Maybe jtype :=
match v with
 | Unit ⇒ Some Void
 | Null ⇒ Some NullType
 | Bool _ ⇒ Some Boolean
 | Intg _ ⇒ Some Integer
 | Str _ ⇒ Some String
```

```
 | Addr _ ⇒ None
end.

Inductive  is_wt_expr : jprog → delta → gamma → expr → jtype → Prop :=
 |  wte_val : ∀ P d g v T,
              typeof v = Some T →
              is_wt_expr P d g (Val v) T
 |  wte_var : ∀ P d g x T,
              wf_gamma P d g →
              g x = Some T →
              is_wt_expr P d g (Var x) T
 ⋮
 |  wte_while : ∀ P d g c e T,
                is_wt_expr P d g c Boolean →
                is_wt_expr P d g e T →
                is_wt_expr P d g (while (c) e) Void
 |  wte_trycatch : ∀ P d g e1 e2 Tex x T,
                   is_wt_expr P d g e1 T →
                   subtype P d (RefType (TC Tex)) TThrowable →
                   is_wt_expr P d (gamma_add g x (RefType (TC Tex))) e2 T →
                   is_wt_expr P d g (TryCatch (e1) Tex x (e2)) T
 ⋮
```

Listing 33: Expression Typing (Excerpt)

3.5.3. Method and Class Typing

The method and class typing in VPJ is based on the corresponding definitions from Variant Parametric Featherweight Java [IV06]. All types used in the declaration of a well-typed method must be well-formed in the type variable environment with method and class type parameters set to be subtypes of their upper bounds (note that methods parameters override class parameters). The method body should be well-typed in the typing environment where the parameters have the corresponding types. Finally, the method declaration must override the same method declared in one of the superclasses (if any) with the same parameter and return types. The override check is done using the auxiliary definition `overrides` defined in Appendix A.4.

```
Inductive  is_wt_method : jprog → cname
                              → mname
                              → Prop :=
 |  wtm_rule : ∀ (J : jprog) d cn mn ctvnts D fns (fd : fdecl) (md : mdecl)
                mtvnts vnts Tbody T ebody
               (Hcn : J cn = Some (ctvnts, D, (fns, fd), md))
               (Hmn : md mn = Some (mtvnts, vnts, T, ebody)),
               d = delta_init ((map (fun x ⇒ (fst x, (JVCov, TC (snd x)))) ctvnts)
                            ++ (map (fun x ⇒ (fst x, (JVCov, snd x))) mtvnts)) →
               (∀ X U, In (X, U) mtvnts →
                       is_wf_type J d (RefType U)) →
```

```
(∀ X U, In (X, U) vnts →
         is_wf_type J d U) →
is_wf_type J d T →
is_wt_expr J d
  (gamma_init ((this, InvClass cn (map (fun x ⇒ TV (fst x)) ctvnts))
               :: vnts)) ebody Tbody →
subtype J d Tbody T →
overrides J D mn (fst (split mtvnts)) (snd (split mtvnts))
                 (snd (split vnts)) T →
  is_wt_method J cn mn.
```

Listing 34: Method Typing

A class declaration is well-typed if all the types used in the declaration are well-formed and all method declarations are well-typed. Finally, we say a VPJ program is well-typed if all class declarations are well-typed.

```
Inductive  is_wt_class : jprog → cname → Prop :=
 |  wtc_rule : ∀ (P : jprog) cn XNs D fns (fd : fdecl) (md : mdecl) d
               (Hcn : P cn = Some (XNs, D, (fns, fd), md)),
              d = delta_init (map (fun x ⇒ (fst x, (JVCov, TC (snd x)))) XNs) →
              (∀ X U,
                 In (X, U) XNs →
                 is_wf_type P d (RefType (TC U))) →
              (∀ vn Tf,
                 fd vn = Some Tf →
                 is_wf_type P d Tf) →
              is_wf_type P d (RefType (TC D)) →
              (∀ mn mtvnts vnts T ebody,
                 md mn = Some (mtvnts, vnts, T, ebody) →
                 is_wt_method P cn mn) →
              is_wt_class P cn.
```

Listing 35: Class Typing

3.6. Single-Threaded Semantics of VPJ

The single-threaded semantics of Variant Parametric Jinja is defined by the reduction relation `red`. The definition of `red` closely follows the analogous relation from Jinja with Threads. We have only changed some of the rules to make them compatible with Coq and removed unneeded rules related to arrays. The complete definition of `red` can be found in Appendix A.5, in this section we explain some of the rules (there are 60 rules in total) to give an idea about VPJ semantics.

The reduction relation is parameterized with the hypothesis `subclass_wf` that is used by field and method lookup functions `fields_of` and `mbody` defined in Appendix A.4 and Appendix A.5 respectively. `red` is a *labeled* relation, it takes a conversion function (see `extTA2J` defined in

Appendix A.5) and reduces some expression together with some state to another expression and a state, possibly producing a *thread action* (usually visualized as a label on the reduction arrow). The thread actions are used for inter-thread communication in multi-threaded semantics, we will explain them in more details in Section 3.7.

As an example we explain how the reduction of the `Call` expression proceeds. A `Call` expression consists of the object expression, type annotations, method name and argument expressions. The object expression is reduced using the reduction rule `CallObj`. Once the object expression is reduced to some value, the arguments can be reduced using the rule `CallArgs`. The actual method call can happen only after all arguments were reduced to values.

There are two kinds of method calls, internal and external. The internal calls are calls to the methods defined in the class of the object we are accessing or one of its superclasses. These calls are executed in the same thread by inlining the method body inside of a sequence of blocks that declare the method parameters, as defined by `RedCall` and the auxiliary function `blocks`. The method body is extracted from the class declaration with the help of function `mbody`. The definitions of these functions can be found in Appendix A.5.

The external calls model the actions that cannot be defined directly using the normal VPJ expressions. This concept is quite flexible, one could add external calls that model features such as file access, networking, inter-process communication etc. We currently use the same external actions as in Jinja with Threads, namely the minimal set of methods needed to support threads and synchronization. Please refer to Section 3.7 for more details.

We also have reduction rules for various error conditions, such as `RedCallNull` which handles the **null** dereference during a method call. Such reduction rules throw a system exception, such as `NullPointer`. Since any subexpression can throw an exception, we also need reduction rules to propagate the exceptions to the parent expressions. An example for such reduction rule is the `CallThrowObj` rule.

Most reduction rules either produce an empty thread action (for example `CastThrow` and `RedCall`) or just pass the thread actions possibly produced by subexpressions (see `CallObj`). However, some expressions need to notify other threads during reduction. For example, when the synchronization block `jsync` is reduced using the rule `LockSynchronized`, it emits a synchronization event and locks the mutex of the corresponding object. The synchronized expression reduces to the `jinsync` block, which means that the lock was taken and the subexpression can now be reduced using other reduction rules.

```
Section  Java.

(* The VPJ program, all definitions below are parameterized by it *)
Variable  P : jprog.
(* Well-formedness assumption *)
Hypothesis  subclass_wf : well_founded (superclass1 P).

Inductive  red : (external_thread_action → J_thread_action) →
                 (expr × (heap × locals)) →
                 J_thread_action →
                 (expr × (heap × locals)) → Prop :=
```

```
| CallObj : ∀ extTA e s ta e' s' tvts M es,
           red extTA (e, s) ta (e', s') →
           red extTA ((Call e tvts M (es)), s) ta
               ((Call e' tvts M (es)), s')
| CallArgs : ∀ extTA es s ta es' s' v M tvts,
            reds extTA es s ta es' s' →
            red extTA ((Call (Val v) tvts M (es)), s) ta
                ((Call (Val v) tvts M (es')), s')
| RedCall : ∀ extTA s a C fds M Ts T D args vns e tvts,
           hp s a = Some (Obj C fds) →
           ¬ is_external_call P (RefType (TC C)) M →
           mbody P subclass_wf C M tvts = Some (D, vns, Ts, T, e) →
           length args = length vns →
           length Ts = length vns →
           red extTA ((Call (jaddr a) tvts M (map Val args)), s) empty_ta
               ((blocks (this :: vns) ((RefType (TC D)) :: Ts)
                        ((Addr a) :: args) e), s)
| RedCallExternal : ∀ extTA s a T M vs va h' ta' ta e',
                    typeof_h (hp s) (Addr a) = Some T →
                    is_external_call P T M →
                    red_external P (hp s) a M vs ta va h' →
                    ta' = extTA ta →
                    e' = extRet2J va →
                    red extTA ((Call (jaddr a) nil M (map Val vs)), s) ta'
                        (e', (h', lcl s))
| RedCallNull : ∀ extTA M vs s tvts,
                red extTA ((Call jnull tvts M (map Val vs)), s) empty_ta
                    ((jTHROW eNullPointer), s)
⋮
| LockSynchronized : ∀ extTA s a arrobj e,
                     hp s a = Some arrobj →
                     red extTA ((jsync (jaddr a) e), s)
                         (ta_update_obs (ta_update_locks empty_ta Lock a)
                                        (Synchronization a)) ((jinsync (a) e), s)
⋮
| CallThrowObj : ∀ extTA a M es s tvts,
                 red extTA ((Call (jThrow a) tvts M (es)), s) empty_ta
                     ((jThrow a), s)
⋮

End Java.
```

Listing 36: Small-Step Semantics (Excerpt)

3.7. Multi-Threaded Semantics of VPJ

The multi-threaded semantics of Variant Parametric Jinja uses the multi-threaded framework developed by A. Lochbihler for Jinja with Threads [Loc08]. We have translated the formalization of this framework to Coq with only minor changes related to the differences between

Isabelle/HOL and Coq. The framework semantics is designed to be very generic. It is parameterized by a (single-threaded) small-step semantics and thus should work the same for our changed reduction relation. In this section, we will give a brief overview of the multi-threaded semantics. Please refer to the corresponding paper for the detailed explanation.

The idea behind the framework semantics is to define an interleaved small-step semantics for the multi-threaded case by using an existing single-threaded small-step semantics as a black box. The multi-threaded state adds additional implicit *locks* to the objects, defines *thread information* (separated stacks etc.) and the *wait sets*. The heap is used as the *shared memory*. The step from the single-threaded to the multi-threaded case is made using the *thread actions*. The thread actions are used for locking, thread creation and manipulation of wait sets.

The framework semantics defines 11 thread actions split into 3 groups. The lock actions are used for locking, unlocking, temporarily releasing and testing whether the monitor has been locked. Thread actions are used to start a new thread (and possibly fail), check if a thread was started and join on a thread. The wait set actions can suspend a thread and notify one or all thread in a wait set. This models the actual semantics of Java threads.

```
Inductive  lock_action : Type :=
 |  Lock
 |  Unlock
 |  UnlockFail
 |  ReleaseAcquire.

(* X = (cname, mname, addr), M = heap *)
Inductive  new_thread_action (X M : Type) : Type :=
 |  NewThread : thread_id → X → M → new_thread_action X M
 |  NewThreadFail : new_thread_action X M
 |  ThreadExists : thread_id → new_thread_action X M.

Inductive  conditional_action : Type :=
 |  Join : thread_id → conditional_action.

Inductive  wait_set_action : Type :=
 |  Suspend : addr → wait_set_action
 |  Notify : addr → wait_set_action
 |  NotifyAll : addr → wait_set_action.

Definition  lock_actions := addr → list lock_action.

(* X = (cname, mname, addr), M = heap, O = (Maybe obs_event) *)
Definition  thread_action (X M O : Type) :=
  lock_actions × (list (new_thread_action X M))
    × (list conditional_action) × (list wait_set_action) × O.

Definition  empty_ta {X M O : Type} : thread_action X M (Maybe O) :=
  (fun x ⇒ nil, nil, nil, nil, None).

Definition  lock := option (thread_id × nat).
Definition  locks := addr → lock.
Definition  released_locks := addr → nat.
Definition  thread_info (X : Type) := thread_id → option (X × released_locks).
```

```
Definition  wait_sets := thread_id → option addr.
Definition  state (X M : Type) := locks × (@thread_info X × M) × wait_sets.

Inductive  obs_event : Type :=
 |  ExternalCall : addr → mname → list val → val → obs_event
 |  Synchronization : addr → obs_event
 |  ThreadStart : thread_id → obs_event
 |  ThreadJoin : thread_id → obs_event.
```

Listing 37: Thread Actions

These actions must of course be supported by the single-threaded small-step semantics. Some reduction rules of the single-threaded reduction relation produce a thread action which is then further reduced by the multi-threaded reduction relation. In our case, there are two cases where they come into play, when the synchronization block is reduced and when an external call is performed. The synchronization block produces a lock action before the subexpression is reduced and an unlock action afterwards. The multi-threaded reduction relation then modifies the lock status of the corresponding object and reduces the expression again using the single-threaded relation. The external calls produce the remaining thread actions and deserve a more closer look.

The External Calls

There are currently 5 external calls, modeling the behavior of the native Java methods Thread.start(), Thread.join(), Object.wait(), Object.notify() and Object.notifyAll(). The decision whether a method is external or not is made during the reduction of the `Call` expression using the function `is_external_call`.

```
Section  Java.

(* Instantiated with expression on use (needed to break circular dependency) *)
Variable  E : Type.

Inductive  is_external_call : @prog E → jtype → mname → Prop :=
 |  ecThreadStart : ∀ P C ps,
                     subclass P C Thread →
                     is_external_call P (Class C ps) start
 |  ecThreadJoin : ∀ P C ps,
                    subclass P C Thread →
                    is_external_call P (Class C ps) join
 |  ecObjectWait : ∀ P T, is_refT T → is_external_call P T wait
 |  ecObjectNotify : ∀ P T, is_refT T → is_external_call P T notify
 |  ecObjectNotifyAll : ∀ P T, is_refT T → is_external_call P T notifyAll.

End Java.
```

Listing 38: Definition of External Calls

The semantics of external calls is defined using the reduction relation `red_external` that is used to reduce the `Call` expression in the `RedCallExternal` rule. The external methods can either succeed, in which case they are reduced to an `external_thread_action`, or fail (with the exception of `join` method), in which case they are reduced to the address of the corresponding system exception (see rules `RedNewThreadFail`, `RedWaitFail` etc.).

The external thread actions produced in case of successful reduction are then further reduced by the multi-threaded reduction relation. The `start` method causes creation of a new thread and schedules its start. The `join` method requests joining some thread. The methods that modify the wait set (`wait`, `notify` and `notifyAll`) first try to acquire a lock on the corresponding thread, to ensure synchronization, unlock the thread again and perform the corresponding action (that is, `Suspend`, `Notify` and `NotifyAll` respectively). Note that the unlock action is performed right after the lock action, this does not cause any race conditions, because the whole action sequence is executed atomically [Loc10].

```
Section Java.

(* Instantiated with expression on use (needed to break circular dependency) *)
Variable E : Type.

Definition  external_thread_action :=
  thread_action (cname × mname × addr) heap (option obs_event).

Inductive  red_external : @prog E → heap → addr → mname →
                          list val → external_thread_action →
                          sum val addr → heap → Prop :=
| RedNewThread : ∀ P h a cn fs,
    h a = Some (Obj (TClass cn nil) fs) →
    subclass P cn Thread →
    red_external P h a start nil
                 (ta_update_obs (ta_update_NewThread empty_ta
                   (NewThread a (cn, run, a) h)) (ThreadStart a))
                 (inl _ Unit) h
| RedNewThreadFail : ∀ P h a cn fs,
    h a = Some (Obj (TClass cn nil) fs) →
    subclass P cn Thread →
    red_external P h a start nil
                 (ta_update_NewThread empty_ta (ThreadExists a))
                 (inr _ (addr_of_sys_exc eIllegalThreadState)) h
| RedJoin : ∀ P h a cn fs,
    h a = Some (Obj (TClass cn nil) fs) →
    subclass P cn Thread →
    red_external P h a join nil
                 (ta_update_obs (ta_update_Conditional empty_ta (Join a))
                                (ThreadJoin a)) (inl _ Unit) h
| RedWait : ∀ P h a,
    red_external P h a wait nil
                 (ta_update_obs (ta_update_locks (ta_update_locks
                   (ta_update_locks (ta_update_wait_set empty_ta (Suspend a))
                     Unlock a) Lock a) ReleaseAcquire a) (Synchronization a))
```

```
                (inl _ Unit) h
| RedWaitFail : ∀ P h a,
    red_external P h a wait nil
                (ta_update_locks empty_ta UnlockFail a)
                (inr _ (addr_of_sys_exc eIllegalMonitorState)) h
| RedNotify : ∀ P h a,
    red_external P h a notify nil
                (ta_update_obs (ta_update_locks (ta_update_locks
                  (ta_update_wait_set empty_ta (Notify a))
                  Unlock a) Lock a) (Synchronization a)) (inl _ Unit) h
| RedNotifyFail : ∀ P h a,
    red_external P h a notify nil
                (ta_update_locks empty_ta UnlockFail a)
                (inr _ (addr_of_sys_exc eIllegalMonitorState)) h
| RedNotifyAll : ∀ P h a,
    red_external P h a notifyAll nil
                (ta_update_obs (ta_update_locks (ta_update_locks
                  (ta_update_wait_set empty_ta (NotifyAll a))
                  Unlock a) Lock a) (Synchronization a)) (inl _ Unit) h
| RedNotifyAllFail : ∀ P h a,
    red_external P h a notifyAll nil
                (ta_update_locks empty_ta UnlockFail a)
                (inr _ (addr_of_sys_exc eIllegalMonitorState)) h.

End Java.
```

Listing 39: Semantics of External Calls

3.8. Symbolic Library

In addition to the standard classes like `Object` required by VPJ we need to define an additional set of classes that can be used by the translated protocols. We call this set the *symbolic library*. The symbolic library has several purposes. First of all, it provides a symbolic abstraction of the cryptographic primitives and channel communication. It is designed to be sound and simple enough to simplify proving the transformation secure. Finally, the symbolic library could also be used for debugging changes to our transformation.

We call this library symbolic, because the implementation of the cryptographic primitives and data types does not perform any "real" cryptography or networking. The objects constructed using the classes of the symbolic library model the Expi terms. Our implementation also has a concrete library that performs "real" cryptographic operations and uses the actual network. The two libraries (concrete and symbolic) can be used interchangeably by the generated code.

The symbolic library consists of 7 classes shown in Figure 3.2. The class `AbstractBase` serves as the base class of the class hierarchy for all data types defined in the library. `AbstractGenerativeBase` is the superclass for generative types. The class hierarchy used for the generated data types is quite flat, each data type class is either a direct subclass of `AbstractBase` or a direct subclass of `AbstractGenerativeBase`.

`Semaphore` implements the semantics of a counting semaphore using the synchronization primitives of VPJ. This class is used in the implementation of inter-thread message passing. Expi channels are modeled using the `AbstractChannel` class. It implements the synchronous semantics of Expi channels, where different processes are implicitly synchronized when a message is sent from one process to another over a channel.

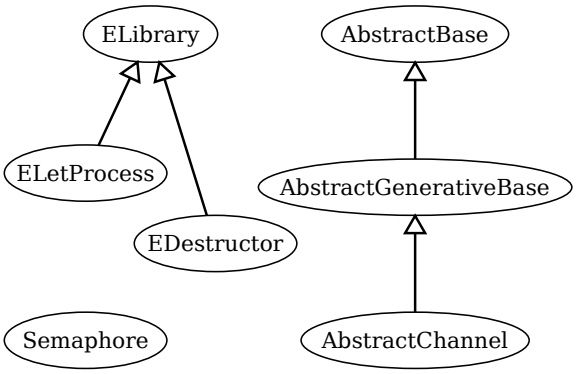


Figure 3.2.: Symbolic Library Classes

Finally, `ELibrary` is the base class for `ELetProcess` and `EDestructor`, the two exception classes used by the transformation from Java to Expi. These exceptions are used to model failing constructor and destructor applications. The translation of the `proc_let` process uses the exception to decide when to take the else branch.

The complete VPJ definitions of the classes in our symbolic library can be found in Appendix A.7. We have proved that all these classes are well-typed with respect to our type system.

4. Formalizing the Transformation from Expi to VPJ

The transformation from Extensible Spi Calculus to Variant Parametric Jinja is performed in two steps. The first step translates the Expi Calculus into the *Global Expi Calculus*, a modification of the source language with a different semantics. The second step finally transforms Global Expi into Variant Parametric Jinja.

4.1. The Global Expi Calculus

The usual semantics of the Expi calculus heavily relies on α-renaming and *scope extrusion* (see the structural equivalence rule `pequiv_scope_extrusion` defined in Section 2.4). Scope extrusion allows to move name binders (in the form of restriction processes `proc_new`) out of a parallel composition (`proc_fork`). The internal reduction relation `red` cannot reduce restriction processes, so in order to reduce a protocol, one has to move all restrictions to the top using structural equivalence and reduce the remaining processes underneath.

The problem with this approach is that this semantics cannot be implemented in mainstream programming languages, because it would require changing the scope of variables and moving them across different processes or even across different machines. Instead, one usually implements restriction processes by generating a globally fresh name and storing the result in a local variable, which eliminates the need for scope extrusion. This solution works well in practice, but makes the gap between the formal model and the implementation even larger.

The resulting differences in the semantics can substantially complicate the reasoning about the transformation from Expi to Java. For example, in order to prove that the transformation preserves some trace properties using bisimulation, we would need to prove that each time the Expi process reduces with some event, the corresponding translation also reduces with the same event (possibly using additional silent steps). Showing this in case when the reduction step on the Expi side includes a scope extrusion is very hard because of the fundamental difference in the semantics mentioned before.

Fortunately, one can give an alternative *global semantics* to the pi calculus, as proposed by L. Wischik [Wis04]. We adapt the idea of global semantics to our setting. The resulting Global Expi Calculus uses the same terms and types as the Extensible Spi Calculus and has only one different process, the generation process `gproc_gen` instead of the restriction process `proc_new`. The other processes stay exactly the same.

```
Section ParametrizedByName.
```

```
Inductive  gproc : Set :=
 |  gproc_out : term → term → gproc → gproc
   (* Binds the message received on the channel in a variable in continuation process *)
 |  gproc_in : term → gproc → gproc
   (* Binds the message received on the channel in a variable in continuation process *)
 |  gproc_bangin : term → gproc → gproc
   (* Binds the result of a successful destructor application in a variable in the first
      continuation process *)
 |  gproc_let : dtor → gproc → gproc → gproc
   (* Binds a fresh Expi name of the given type in continuation process *)
 |  gproc_gen : type → gproc → gproc
 |  gproc_fork : gproc → gproc → gproc
 |  gproc_null : gproc.

End ParametrizedByName.
```

Listing 40: Processes in Global Expi

The main difference to the Expi Calculus is in the semantics of internal reduction and structural equivalence. The structural equivalence relation `gequiv` is much simpler than the structural equivalence used in the Expi Calculus. The relation is symmetric, reflexive and transitive and also keeps the two rules stating the commutativity and associativity of parallel composition, but omits the problematic scope extrusion rule.

The reduction relation in the Global Expi Calculus has the same rules for reducing the let process and communication using input and output processes, but has no rule to reduce underneath restriction processes like `red_new` in `red`. Instead, it defines a way to directly reduce the generation process `gproc_gen` using the rule `gred_gen`. The semantics of `gproc_gen` models the implementation of the restriction process in a mainstream programming language, it generates a globally fresh Expi name and substitutes it for occurrences of the variable that was bound by the generation process.

The rule `red_fork` is not explicitly present in `gred`, it is merged into the other rules (where applicable) in the form of an additional contextual process (usually called `F`). The reason for that is the "global freshness" condition in `gproc_gen` – we cannot allow to reduce subprocesses independently from each other, because the reduction could introduce free names that are already used in other subprocesses.

```
Section ParametrizedByName.

Inductive  gequiv : gproc → gproc → Prop :=
 |  gequiv_null : ∀ (G : gproc),
     lc_gproc G →
     gequiv (gproc_fork G gproc_null) G
 |  gequiv_comm : ∀ (G H : gproc),
     lc_gproc H →
     lc_gproc G →
     gequiv (gproc_fork G H) (gproc_fork H G)
 |  gequiv_assoc : ∀ (G H F : gproc),
     lc_gproc G →
```

```
    lc_gproc H →
    lc_gproc F →
    gequiv (gproc_fork (gproc_fork G H) F) (gproc_fork G (gproc_fork H F))
| gequiv_refl : ∀ (G : gproc),
    lc_gproc G →
    gequiv G G
| gequiv_symm : ∀ (G H : gproc),
    gequiv H G →
    gequiv G H
| gequiv_trans : ∀ (G F H : gproc),
    gequiv G H →
    gequiv H F →
    gequiv G F
| gequiv_fork : ∀ (G F H : gproc),
    lc_gproc F →
    gequiv G H →
    gequiv (gproc_fork G F) (gproc_fork H F).

Inductive  gred : gproc → gproc → Prop :=
| gred_gen : ∀ (T : type) (G F : gproc) (a : atom),
    (a ∉ (fn_in_gproc (gproc_fork (gproc_gen T G) F))) →
    gred (gproc_fork (gproc_gen T G) F)
         (gproc_fork (open_gproc_wrt_term G (term_Nam (Nam_f a))) F)
| gred_io : ∀ (c : atom) (t : term) (G H F : gproc),
    lc_gproc (gproc_in (term_Nam (Nam_f c)) H) →
    lc_gproc G →
    lc_term t →
    lc_gproc F →
    gred (gproc_fork (gproc_out (term_Nam (Nam_f c)) t G)
                     (gproc_fork (gproc_in (term_Nam (Nam_f c)) H) F))
         (gproc_fork G (gproc_fork (open_gproc_wrt_term H t) F))
| gred_bangio : ∀ (c : atom) (t : term) (G H F : gproc),
    lc_gproc G →
    lc_term t →
    lc_gproc (gproc_bangin (term_Nam (Nam_f c)) H) →
    lc_gproc F →
    gred (gproc_fork (gproc_out (term_Nam (Nam_f c)) t G)
                     (gproc_fork (gproc_bangin (term_Nam (Nam_f c)) H) F))
         (gproc_fork G (gproc_fork (open_gproc_wrt_term H t) (gproc_fork
                                    (gproc_bangin (term_Nam (Nam_f c)) H) F)))
| gred_dtor : ∀ (g : dtor) (G H F : gproc) (t : term),
    lc_gproc (gproc_let g G H) →
    lc_gproc H →
    lc_gproc F →
    g_red C g = Some t →
    lc_term t →
    gred (gproc_fork (gproc_let g G H) F) (gproc_fork (open_gproc_wrt_term G t) F)
| gred_else : ∀ (g : dtor) (G H F : gproc),
    lc_gproc (gproc_let g G H) →
    lc_gproc H →
    lc_gproc F →
    g_red C g = None →
```

```
    gred (gproc_fork (gproc_let g G H) F) (gproc_fork H F)
| gred_equiv : ∀ (G H G' H' : gproc),
    gequiv G G' →
    gred G' H' →
    gequiv H' H →
    gred G H.

End ParametrizedByName.
```

Listing 41: Semantics of Global Expi Calculus

In the setting of the pi calculus, the global semantics was shown to be equivalent to the usual scope-extrusion-based semantics [Wis04]. This makes the Global Expi Calculus a perfect intermediate language for our transformation. This allows us to completely avoid dealing with scope extrusion when transforming Global Expi to VPJ.

4.2. First Step: Expi to Global Expi

The transformation from Extensible Spi Calculus to Global Expi Calculus is very simple. Both calculi use the same terms, types and configuration, we just use them as they are. The processes are transformed using the relation `translate1`. In most cases, the processes just need to be renamed (see Appendix A.9 for definitions of the corresponding rules). The only interesting case is the transformation of the restriction process `proc_new` to the generation process `gproc_gen`. In this case we just substitute the name used by the restriction process with a fresh variable for the generation process.

```
Section ParametrizedByName.

Inductive  translate1 : proc → gproc → Prop :=
| trans1_new : ∀ (L : vars) (T : type) (P : proc) (G : gproc),
    (∀ x, x ∉ L →
       (∀ a, a ∉ L u {{ x }} →
          translate1 (subst_nam_with_term_in_proc (term_Var_f x) (Nam_f a)
                        (open_proc_wrt_nam P (Nam_f a)))
                     (open_gproc_wrt_term G (term_Var_f x)))) →
    translate1 (proc_new T P) (gproc_gen T G)
⋮

End ParametrizedByName.
```

Listing 42: Transformation from Expi to Global Expi

4.3. Second Step: Global Expi to VPJ

In the second step of the transformation we have to implement the behavior of Expi terms and processes in VPJ, which requires a substantial amount of code. This makes the second step of our transformation much more challenging than the first.

Table 4.1 shows an overview of the different language constructs we need to transform and gives a rough idea how they are represented in VPJ. Expi types are modeled as additional classes in our symbolic library. We use variant parametric types to represent Expi type parameters. The instances of the classes corresponding to Expi types represent Expi terms of the corresponding type. The configuration names are used in the destructor reduction relation, we store them in fields of type `String` in the classes representing Expi types. Expi constructors and destructors are represented as special methods in a symbolic library class named `Fun`. We use a simple naming convention to distinguish constructor and destructor methods. The terms are represented by VPJ expressions that either access local variables or call constructor methods. The processes are transformed into larger code blocks that create and modify the terms stored in local variables and use the symbolic library for interaction with each other. We use threads to model parallel composition of processes and shared memory to pass data between them.

Global Expi		**VPJ**
Configuration:		
- Type identifiers (`t_idents`)	$\rightsquigarrow$	Class declarations
- Configuration names (`cfg_name`)	$\rightsquigarrow$	String constants
- Constructors	$\rightsquigarrow$	Special methods in the class `Fun`
- Destructors	$\rightsquigarrow$	Special methods in the class `Fun`
Terms	$\rightsquigarrow$	Expressions (variables, method calls)
Processes	$\rightsquigarrow$	Expressions (variable declarations, control flow)
- Parallel composition (`gproc_fork`)	$\rightsquigarrow$	Threads
Free names	$\rightsquigarrow$	Variables in main (passed by reference to threads)

Table 4.1.: Transformation Overview

In the following sections we explain each part of the transformation in more detail and present the most important definitions. All transformation functions are enclosed into a Coq section `ParametrizedGlobalPi` and parameterized with an injective function `atos` that translates an `atom` to `string`, the set of identifiers `Idents` and an injective function `idtostr` that translates values of type `Idents` to `string`. We also use a large number of small helper functions that construct various lists, generate fresh identifiers etc. For readability reasons, we will only briefly describe their functionality when needed, please refer to Appendix A.10 for the corresponding definitions. The symbolic library required for the resulting code was presented in Section 3.8.

```
Section ParametrizedGlobalPi.


(* Injective function that converts atom to string *)
Variable atos : atom → string.
Hypothesis atos_inj : ∀ x y, atos x = atos y → x = y.

(* A common set of Expi identifiers *)
Variable Idents : Set.
(* Injective function that converts Idents to string *)
```

```
Variable  idtostr : Idents → string.
Hypothesis  idtostr_inj : ∀ x y, idtostr x = idtostr y → x = y.

...

End ParametrizedGlobalPi.
```

Listing 43: Transformation Assumptions

4.3.1. Classes Representing Expi Types

Expi types are transformed into VPJ class declarations using the relation `spitoj_type_class`. The name of the resulting class is obtained using the function `idtostr`. We use either `AbstractGenerativeBase` or `AbstractBase` as the superclass, depending on whether the type is generative or not. The resulting flat class hierarchy reflects the subtyping relation in the Expi calculus. The class `AbstractBase` represents the top type, its subclasses represent the other Expi types. Variant parametric VPJ types provide a precise model of the parametric polymorphism we have in Expi. Expi type parameters are encoded as VPJ type parameters, with names generated with the auxiliary function `fresh_tvname_list` that produces a list of different type variable names of the form `"a"`, `"aa"`, `"aaa"` etc. The actual class declaration is generated using the function `new_type_class`.

Instances of the classes representing Expi types are used to represent terms. In Global Expi, terms can be created in two ways, either generated by the process `gproc_gen` or constructed out of other terms using an Expi constructor. We need to preserve the structure of the terms in our model to be able to access subterms in destructors. In particular, we store the name of the constructor that was used to create the term in a field named `"ctorName"` of type `String` (defined in `AbstractBase`). This field is set to `jnull` if the term is created using the generation process. Furthermore, to preserve the term structure we also store the arguments given to the constructor used to create the term in fields. We define a list of fields for all arguments of all Expi constructors that return the corresponding type in the declaration of the generated classes. The list of such constructors together with their argument types is produced by the helper function `ctors_for_type`. Another auxiliary function `ctor_args` transforms this list into a list of field name and field type pairs that is used to initialize the functional map defining the fields of the generated class.

Each class representing an Expi type has one method called `"equals"`. This method is inspired by the standard Java method of the same name, it takes one argument of the same type as the class where it is declared and returns a `Boolean`. The implementation of `"equals"` is defined by `type_mdef_equals` and `compare_ctor_args`. These two functions implement structural equality on Expi terms by comparing the constructor names of the terms and recursively comparing their arguments. We use pointer equality to compare terms created by the generation process.

```
Section ParametrizedGlobalPi.
```

```
Fixpoint  compare_ctor_args (cn : cname) (b : expr)
                            (cargs : list (vname × jtype)) : expr :=
match cargs with
 | nil ⇒ b
 | (x, RefType (TC T)) :: xs ⇒
     (* if (this.x == null) *)
        (:if ((FAcc jthis x cn) == jnull) :then (
         (* (this.x == obj.x) && ...  *)
            (FAcc jthis x cn) == (FAcc (Var "obj") x cn)
        ) :else (
         (* this.x.equals(obj.x) && ...  *)
            Call (FAcc jthis x cn) nil "equals" ((FAcc (Var "obj") x cn) :: nil)
        )) && (compare_ctor_args cn b xs)
 | (x, T) :: xs ⇒
     (* (this.x == obj.x) && ...  *)
        ((FAcc jthis x cn) == (FAcc (Var "obj") x cn))
             && (compare_ctor_args cn b xs)
end.

Definition  type_mdef_equals cn tvns cargs : jmdef :=
  (nil, ("obj", Class cn (map (fun tvn ⇒ (JVInv, TV tvn)) tvns)) :: nil, Boolean,
    (* if (this == obj) *)
       :if (jthis == (Var "obj")) :then (
         (* return true; *)
           jtrue
    (* else if (obj == null) *)
       ) :else (:if ((Var "obj") == jnull) :then (
         (* return false; *)
           jfalse
       ) :else (
         (* ...  && (this.ctorName == obj.ctorName) *)
           compare_ctor_args cn
             ((FAcc jthis "ctorName" cn) == (FAcc (Var "obj") "ctorName" cn))
             cargs
       ))).

Definition  new_type_class (cn : cname) (tvns : list tvname) (sn : cname)
                           (ctrs : list (string × list jtype)) : jclass :=
let cargs := (ctor_args ctrs) in
  (map (fun tvn ⇒ (tvn, CAbstractBase)) tvns,
   TClass sn nil,
   (map fst cargs, fields_init cargs),
   methods_init ( (* boolean equals(TypeName obj) *)
                 ("equals", type_mdef_equals cn tvns cargs) :: nil)).

Inductive  spitoj_type_class : @config Idents →
                               Idents →
                               cname →
                               jprog →
                               Prop :=
 |  spitoj_type_class_ngen :
     ∀ tn c J cn vs,
       cn = idtostr tn →
```

```
      t_varmap c tn = Some vs →
      is_t_gen c (type_Nested "" tn nil) = false →
      J cn = Some (new_type_class cn (fresh_tvname_list (length vs))
                                     "AbstractBase"
                                     (ctors_for_type c tn (f_idents c))) →
      spitoj_type_class c tn cn J
| spitoj_type_class_gen :
    ∀ tn c J cn vs,
      cn = idtostr tn →
      t_varmap c tn = Some vs →
      is_t_gen c (type_Nested "" tn nil) = true →
      J cn = Some (new_type_class cn (fresh_tvname_list (length vs))
                                     "AbstractGenerativeBase"
                                     (ctors_for_type c tn (f_idents c))) →
      spitoj_type_class c tn cn J.

End ParametrizedGlobalPi.
```

Listing 44: Classes representing Expi types

When an Expi type is used, for example in a type annotation or in the restriction process, it is transformed into the corresponding VPJ class type (`jclass_type`). The top type `type_Top` is transformed into `TCAbstractBase`. Channel type `type_Channel` is transformed into the parametric type `TCAbstractChannel` instantiated with the transformation of the type parameter. Custom types `type_Nested` are transformed into corresponding classes.

Transforming the variances of parametric types is a bit tricky. The type systems of the Expi calculus and VPJ both support parametric types with variances, but use two completely different approaches to define *which* variance should be used. In the Expi calculus, the variance of each type parameter is fixed by the definition of the type. VPJ uses a more flexible approach, here the class declaration only declares the names of type parameters and the variance is chosen when the class is *used*, i.e., in the `New` expression or in the declaration of method parameters or local variables. In both cases, the variance has a similar effect on the subtyping relation.

We transform the variances in the way that makes the VPJ type system behave like the Expi type system. In particular, there are only a few cases where the subtyping plays a role in Expi calculus – when a term is sent over a channel and when it is used in a constructor or destructor application. We can ignore the case with the channel, because the channel type is invariant. This leads to the following strategy: we transform all types used in the processes as invariant and only use variances in the parameter declarations of the Expi constructor and destructor methods. This is achieved simply by using two slightly different functions, `spitoj_type` and `spitoj_type_v` (plus the corresponding list versions). The variant version `spitoj_type_v` additionally takes the variance for each type parameter from the configuration and transforms it into the corresponding VPJ variant. Expi type variables are transformed into VPJ type variables with the same name.

```
Section ParametrizedGlobalPi.
```

```
Fixpoint  spitoj_type (t : @type Idents) : jparam_type :=
match t with
 | type_Channel _ t ⇒ TCAbstractChannel (JVInv, spitoj_type t)
 | type_Nested _ tn ts ⇒ TC (InvTClass (idtostr tn) (map spitoj_type ts))
 | _ ⇒ TCAbstractBase
end.
Definition  spitoj_type_list (ts : list (@type Idents)) : list jparam_type :=
  map spitoj_type ts.

Definition  spitoj_var (v : variance) : jvariance :=
match v with
 | VCo ⇒ JVCov
 | VContra ⇒ JVCon
 | VIn ⇒ JVInv
end.


(* additionally translates variant parameters *)
Fixpoint  spitoj_type_v (c : @config Idents) (t : @type Idents) : jparam_type :=
match t with
 | type_Channel _ t ⇒ TCAbstractChannel (JVInv, spitoj_type_v c t)
 | type_Nested _ tn ts ⇒
     match t_varmap c tn with
      | Some vs ⇒ TC (TClass (idtostr tn) (combine (map spitoj_var vs)
                                                   (map (spitoj_type_v c) ts)))
      | _ ⇒ TCAbstractBase
     end
 | type_Top ⇒ TCAbstractBase
 | type_Var s ⇒ TV s
end.
Definition  spitoj_type_v_list (c : @config Idents) (ts : list (@type Idents))
                               : list jparam_type :=
  map (spitoj_type_v c) ts.

End ParametrizedGlobalPi.
```

Listing 45: Transforming Types

4.3.2. Expi Constructor Methods

Expi constructors are transformed into special methods of the class `Fun`. This class has no other purpose but being a container for Expi constructor and destructor methods. The function `ctor_method` defines the transformation for Expi constructors. If the constructor has a parametric type, the resulting method declaration is parameterized by type variables with `AbstractBase` (the top type) as their upper bounds.

We pass the configuration name used in the constructor application as the first argument to the method. The actual constructor arguments are transformed using the auxiliary function `ctor_params` that produces variable names of form `"a"`, `"aa"`, `"aaa"` etc. The argument types may contain variant type parameters, we need to use the function `spitoj_type_v_list` to

transform them (it also translates variances and type parameters, see Section 4.3.1 for more details). The other uses of types are transformed as usual.

The body of an Expi constructor method simply creates an instance of its return type and initializes all fields using the corresponding values, i.e., the passed configuration name, constructor name and the remaining arguments. As discussed in Section 4.3.1, a class corresponding to an Expi type has a list of fields for each argument of each Expi constructor that can be used to create a term of that type. We use a helper function `ctor_pass_params` to create a list of arguments that pass the corresponding Expi constructor method parameters at the right positions and `jnull` everywhere else.

```
Section ParametrizedGlobalPi.

Definition  ctor_mname (cn : string) : mname :=
  String.append cn "Ctor".

Definition  ctor_method (c : @config Idents) (fn : Idents) : option jmdef :=
match (f_type c "" fn) with
 | None ⇒ None
 | Some (ftype Xs Ts T) ⇒
     match T, (spitoj_type T) with
      | type_Nested _ tn _, TC (TClass cn cps) ⇒
        Some ( (* type variable declarations *)
              map (fun s ⇒ (s, TCAbstractBase)) Xs,
              (* argument names and their types *)
              ("configName", String) :: ctor_params (spitoj_type_v_list c Ts),
              (* return type *)
              RefType (TC (TClass cn cps)),
              (* body expression *)
           (* T ret; *)
              Block "ret" (RefType (TC (TClass cn cps))) None (
           (* ret = new T(configName, fn, a, aa, ..., null, ...); *)
              "ret" ::= New (TClass cn cps)
                            ((Var "configName") :: (Val (Str (idtostr fn))) ::
                             (ctor_pass_params (idtostr fn)
                                 (ctors_for_type c tn (f_idents c)))) ;
           (* return ret; *)
              (Var "ret")))
      | _, _ ⇒ None
     end
end.

End ParametrizedGlobalPi.
```

Listing 46: Expi Contructor Methods

4.3.3. Expi Destructor Methods

Similarly to the Expi constructors, the Expi destructors are also transformed into parameterized methods of the class `Fun`. The most interesting part of the generated destructors

methods is the body expression generated by the function `code_reduction_rules`. The destructor reduction relation in the Expi calculus works by pattern matching the arguments given to the destructor against the reduction rules and constructing the resulting term from the matched terms and constructor applications. In case multiple reduction rules apply, one of them is chosen non-deterministically.

Unfortunately VPJ does not support this form of pattern matching, so we need to implement it by hand. We do this by generating the code for destructor methods that performs the pattern matching for the corresponding reduction rule at runtime. This approach leads to a simpler implementation in comparison to a generic VPJ method that can perform pattern matching for any given reduction rule. Nonetheless, the implementation of the transformation is quite large, we will only explain the most important steps. Our implementation is fully deterministic, we always take the first matching reduction rule. This is an important limitation caused by the fact that VPJ currently lacks convenient ways to nondeterministically vary control flow.

Our pattern matching algorithm is repeated for each reduction rule until one of them applies (see `code_reduction`). We throw an exception to abort execution of the destructor method in case that the matching fails at some point. The reason for using exception handling for this task is mainly the simplification of the resulting implementation.

The implementation of pattern matching consists of 3 main steps. First, we compare the structure of each method argument and the corresponding reduction rule argument. We compare constructor and configuration names stored in the objects and abort if one of the terms does not match. Then, we build a list of "access expressions" that can be used to access the subterms matched by the variables contained in the reduction rule. Finally, we compare the corresponding variables with each other using their `"equals"` methods and, provided all equality checks succeeded, construct the return term using constructor method calls and the matched variables.

The code for the last step is generated using the function `code_var_match`. This last step requires a small trick, we need to store *each* matched subterm in a different local variable to be able to compare it with others and use it in the return term. In particular, we need to know how many variables are used in a reduction rule and know how to deduce their names. The simplest way to achieve that was to require the reduction rules to provide that number and use bound variables for the variables.

```
Section ParametrizedGlobalPi.

Definition  dtor_mname (gn : string) : mname :=
  String.append gn "Dtor".

Fixpoint  code_var_match (c : @config Idents) (gn : Idents) (Tret : jparam_type)
                         (n : nat) (vlist : list (nat × expr × jtype))
                         (x : @term Idents) : expr :=
match n with
       (* return ...; *)
 | O ⇒ code_return_term c x Tret
 | S m ⇒ let v := (nat_to_vname m) in
```

```
          match (project_n m vlist) with
           | (path1, T) :: vnlist ⇒
              (* T v; *)
                 Block v T None (
              (* v = path1; *)
                 v ::= path1 ;
              (* if (v == null || !(v.equals(path1) && ...  ))  *)
                 :if (:if ((Var v) == jnull) :then jtrue
                      :else (jnot (code_compare_vars (Var v) vnlist))) :then (
                  (* throw new EDestructor(); // match failed *)
                     throw (New CEDestructor nil) ;
                     Cast (RefType Tret) jnull
                 ) :else (
                    (* recurs *)
                     code_var_match c gn Tret m vlist x
                 ))
           | _ ⇒ code_var_match c gn Tret m vlist x
          end
end.

Definition  code_reduction (c : @config Idents) (gn : Idents) (T : jparam_type)
                           (n : nat) (xs : list (@term Idents))
                           (Ts : list (@type Idents)) (x : @term Idents)
                           (eelse : expr) : expr :=
    (* 1.  match constructors *)
    :if (code_match_ctors c gn xs) :then (
        (* 2.  build the list of matched subterms *)
        let vlist := (code_access_vars c gn xs Ts) in
        (* 3.  compare corresponding variables and construct the return term *)
        code_var_match c gn T n vlist x
    ) :else (
        eelse
    ).

Fixpoint  code_reduction_rules (c : @config Idents) (gn : Idents) (T : jparam_type)
                               (Ts : list (@type Idents)) (rules : list (nat
                                  × list (@term Idents) × @term Idents)) : expr :=
match rules with
 | nil ⇒ throw (New CEDestructor nil) ;
                       Cast (RefType T) jnull
 | (n, xs, x) :: rs ⇒ code_reduction c gn T n xs Ts x
                                      (code_reduction_rules c gn T Ts rs)
end.

Definition  dtor_method (c : @config Idents) (gn : Idents) : option jmdef :=
match (g_type c "" gn) with
 | None ⇒ None
 | Some (ftype Xs Ts T) ⇒
     let Tret := (spitoj_type T) in
     Some ( (* type variables declarations *)
            map (fun s ⇒ (s, TCAbstractBase)) Xs,
            (* argument names and their types *)
            ("configName", String) :: ctor_params (spitoj_type_v_list c Ts),
```

```
            (* return type *)
            RefType Tret,
            (* body expression *)
            code_reduction_rules c gn Tret Ts (g_rules c "" gn))
end.

End ParametrizedGlobalPi.
```

Listing 47: Expi Destructor Methods

The implementation of pattern matching uses many helper functions. `code_match_ctor` generates a boolean expression to recursively compare a destructor parameter with the corresponding argument from a reduction rule using constructor and configuration names stored in the objects. `code_access_var` generates a list of (index, access expression, corresponding type) triples used to access the subterms of constructor applications. `project_n` projects a list of (access expression, expression type) pairs from the list of triples generated by `code_access_var` for some given index `n`. `jnot` produces a code that negates the given boolean expression. `code_return_term` essentially just transforms the given Expi term to VPJ using `spitoj_term`. `code_compare_vars` generates a big conjunction that compares all given expressions with each other by calling their `"equals"` methods. Please refer to Appendix A.10 for the definitions of these functions

4.3.4. Terms

We transform Expi terms into instances of the classes representing Expi types using the function `spitoj_term`. Expi names and variables are represented by local variables of the corresponding type. These local variables are declared by the transformations of the corresponding processes (see Section 4.3.5). In general, we can only transform the named versions of the terms (we need to know what variable name to use), therefore, one has to open locally-closed terms before transformation. An exception is the definition of destructor reduction rules, here we use bound variables `term_Var_b` to simplify the implementation of reduction rules (see Section 4.3.3 for details). Constructor applications are transformed into calls to the corresponding constructor method using a temporary instance of the `Fun` class (like Jinja, VPJ does not have static methods).

`spitoj_term` is a partial function, we implement it using the error monad `Maybe`. We use several error monad functions defined in Appendix A.2 to simplify the definition. `bind` simply applies the given function (first argument) to the second argument if it is not `None`. `option_map` is a standard Coq function that does the same, but needs a function of a slightly different type. `unmaybe` transforms a list that does not contain `None` into a list of values. `instantiate` instantiates the functional type, it is defined in Section 2.6.

```
Section ParametrizedGlobalPi.

Definition  Fun := "Fun".
Definition  CFun : jclass_type := TClass Fun nil.
Definition  TFun : jtype := Class Fun nil.
```

```
Definition  ctor_app (fn : string) (cfg : string) (As : list jparam_type)
                     (es : list expr) : expr :=
  Call (New CFun nil) As (ctor_mname fn) (Val (Str cfg) :: Val (Str fn) :: es).

Fixpoint  spitoj_term (c : @config Idents) (g : gamma) (x : @term Idents)
                      {struct x} : Maybe (expr × jtype) :=
match x with
 | term_Nam (Nam_f v) ⇒ bind (fun T ⇒ Some (Var (atos v), T)) (g (atos v))
 | term_Var_f v ⇒ bind (fun T ⇒ Some (Var (atos v), T)) (g (atos v))
 | term_Ctor cfg fn As xs ⇒
     match unmaybe (map (fun p ⇒ option_map fst (spitoj_term c g p)) xs),
               instantiate As (f_type c cfg fn) with
      | Some es, Some (Tss, Tsx) ⇒ Some (ctor_app (idtostr fn) cfg
                                                  (spitoj_type_list As) es,
                                         RefType (spitoj_type Tsx))
      | _, _ ⇒ None
     end
   (* needed to translate destructor rules *)
 | term_Var_b n ⇒ bind (fun T ⇒ Some (Var (nat_to_vname n), T))
                       (g (nat_to_vname n))
 | _ ⇒ None
end.
Definition  spitoj_term_list (c : @config Idents) (g : gamma)
                             (xs : list (@term Idents)) : Maybe (list expr) :=
  unmaybe (map (fun p ⇒ option_map fst (spitoj_term c g p)) xs).

End ParametrizedGlobalPi.
```

Listing 48: Transforming Terms

4.3.5. Processes

Processes are transformed using the relation `spitoj_gproc`. We take a Global Expi process and transform it into a VPJ program (i.e., class declarations) and an expression. Additionally, we use several lists to keep track of used Expi names and class names (to ensure freshness of all generated names) and a list of currently visible variables together with their types (needed for transformation of the parallel composition, we will explain it below).

In most cases, the transformation is quite obvious, all subprocesses, subterms and types are transformed using the corresponding functions and no new class declarations are generated. The `gproc_null` process is transformed into a "do-nothing" operation. Process `gproc_out` calls the `"send"` method on the channel variable and gives it the transformation of the term as an argument (see `out_expr`). The `gproc_in` process declares a new local variable and assigns the result of a call to `"receive"` on the channel variable to that new local variable (see `in_expr`). The Expi semantics of `gproc_bangin` is to spawn a new copy of the continuation process whenever a message is received on the channel. We implement this semantics using an endless loop. We block on the channel trying to receive some data and start a new thread with the continuation process as soon as we receive something (see `bangin_expr`). The `gproc_let` process is transformed into a try-catch block where we declare a new local variable, assign it

the result of the destructor application (transformed into the destructor method call) and proceed with the else-branch if an ELetProcess exception is caught (see let_expr).

```
Section ParametrizedGlobalPi.

Definition  out_expr (c : vname) (et : expr) (eg : expr) : expr :=
 (* c.send(et) ; *)
    Call (Var c) nil "send" (et :: nil) ;
 (* G *)
    eg.

Definition  let_expr (x : vname) (Tx : jclass_type) (gn : string) (cfg : string)
                     (As : list jparam_type) (es : list expr)
                     (eg eh : expr) : expr :=
  :try :{
   (* T x; *)
      Block x (RefType (TC Tx)) None (
       (* x = x1.gnDtor(x2, ...); *)
          x ::= Call (New CFun nil) As (dtor_mname gn) (Val (Str cfg) :: es) ;
       (* G *)
          eg
      )
   (* catch (ELetProcess ex) *)
  :} :catch CELetProcess :ex (String.append "e" x) :{
   (* H *)
      eh
  :}.

Definition  in_expr (x : vname) (Tx : jclass_type) (vc : vname) (eg : expr) : expr :=
  (* T x; *)
     Block x (RefType (TC Tx)) None (
      (* x = c.receive(); *)
         x ::= Call (Var vc) nil "receive" (nil) ;
      (* G *)
         eg
     ).

Definition  bangin_expr (x : vname) (Tx : jclass_type) (cn : vname)
                        (eg : expr) : expr :=
  while (jtrue) (
      in_expr x Tx cn eg
  ).

End ParametrizedGlobalPi.
```

Listing 49: Transforming Processes

The transformation of the gproc_gen process also declares a new local variable, similar to the gproc_in process and stores a newly generated instance of the corresponding type in it (see gen_expr). However, the corresponding transformation rule spitoj_gproc_gen is a bit more complicated, we need to distinguish between type_Channel and type_Nested in order to know which configuration name to use to initialize the fields of the new object. Also, in cases

gproc_gen, gproc_let, gproc_in and gproc_bangin we need to update the list of local variables, because we declare a new one.

```
Section ParametrizedGlobalPi.

Definition  gen_expr (x : vname) (T : jclass_type) (cfg : string)
                     (cargs : list (vname × jtype)) (eg : expr) : expr :=
  let (cn, ps) := T in
  (* T x; *)
     Block x (RefType (TC T)) None (
      (* x = new T(cfg, null); *)
         x ::= New (TClass cn ps)
                   (Val (Str cfg) :: jnull :: (map (fun x ⇒ jnull) cargs)) ;
      (* G *)
         eg
     ).

End ParametrizedGlobalPi.
```

Listing 50: Transforming Generation Process

Finally, gproc_fork has the most complicated transformation. We model parallel composition using threads, so we need to declare two new thread classes as defined by new_thread. In the target expression we start both threads and call "join" on them (see fork_expr). The resulting thread classes are non-parameterized direct subclasses of Thread and override the method "run". We copy all currently visible local variables into each thread and store them in fields with the same name to mimic the semantics of the Expi calculus, where all previously defined Expi names and variables are visible in all subprocesses. We copy these fields into local variables declared in the body of the "run" method using the helper function localize_fields. This step is needed, because in VPJ we have different expressions for accessing local variables and fields, and the transformation function spitoj_term uses the expression Var to read local variables.

```
Section ParametrizedGlobalPi.

Definition  fork_expr (G H : cname) (Gvn Hvn : vname) (args : list expr) : expr :=
 (* ThreadG Gvar = new ThreadG(); *)
    Block Gvn (Class G nil) None (
    Gvn ::= New (TClass G nil) args ;
 (* ThreadH Hvar = new ThreadH(); *)
    Block Hvn (Class H nil) None (
        Hvn ::= New (TClass H nil) args ;
     (* Gvar.start(); *)
        Call (Var Gvn) nil start nil ;
     (* Hvar.start(); *)
        Call (Var Hvn) nil start nil ;
     (* Gvar.join(); *)
        Call (Var Gvn) nil join nil ;
     (* Hvar.join(); *)
```

```
        Call (Var Hvn) nil join nil
    )).

Definition  new_thread (cn : cname) (xs : list vname) (Ts : list jtype)
                       (e : expr) : jclass :=
  (nil,
   TClass Thread nil,
   (xs, declare_fields xs Ts),
   methods_init ( (* void run() *)
                 (run, (nil, nil, Void,
                        localize_fields cn xs Ts (
                            :try :{
                                e
                            :} :catch CELibrary :ex "x" :{
                                NOP
                            :}))) :: nil)).

Inductive  spitoj_gproc : @config Idents →
                          list cname →
                          list atom →
                          list vname →
                          gamma →
                          @gproc Idents →
                          jprog →
                          expr →
                          Prop :=
 |  spitoj_gproc_out :
     ∀ c cns L vns g ch t G cn et Tt Tt' J eg,
       lc_term ch →
       spitoj_term c g ch = Some (Var cn, TAbstractChannel (JVInv, Tt')) →
       lc_term t →
       spitoj_term c g t = Some (et, Tt) →
       spitoj_gproc c cns L vns g G J eg →
       spitoj_gproc c cns L vns g (:out( ch , t );; G) J (out_expr cn et eg)
 |  spitoj_gproc_in :
     ∀ c cns L vns g ch G cn a x Tx J eg,
       lc_term ch →
       spitoj_term c g ch = Some (Var cn, TAbstractChannel (JVInv, TC Tx)) →
       (∀ x, ¬ In x L → lc_gproc (open_gproc_wrt_term G (term_Var_f x))) →
       a = fresh_atom L →
       x = (atos a) →
       spitoj_gproc c cns (a :: L) (x :: vns) (gamma_add g x (RefType (TC Tx)))
                    (open_gproc_wrt_term G (term_Var_f a)) J eg →
       spitoj_gproc c cns L vns g (:in( ch );; G) J (in_expr x Tx cn eg)
 |  spitoj_gproc_bangin :
     ∀ c cns L vns g ch G cn a x Tx J eg,
       lc_term ch →
       spitoj_term c g ch = Some (Var cn, TAbstractChannel (JVInv, TC Tx)) →
       (∀ x, ¬ In x L → lc_gproc (open_gproc_wrt_term G (term_Var_f x))) →
       a = fresh_atom L →
       x = (atos a) →
       spitoj_gproc c cns (a :: L) (x :: vns) (gamma_add g x (RefType (TC Tx)))
```

```
                (open_gproc_wrt_term G (term_Var_f a)) J eg →
      spitoj_gproc c cns L vns g (!in( ch );; G) J (bangin_expr x Tx cn eg)
| spitoj_gproc_let :
    ∀ c cns L vns g cfg gx As ts es gn a x Tx G H J eg eh Tss Tsx Ajs,
      lc_dtor (dtor_Dtor cfg gx As ts) →
      spitoj_term_list c g ts = Some es →
      gn = idtostr gx →
      (∀ x, ¬ In x L → lc_gproc (open_gproc_wrt_term G (term_Var_f x))) →
      a = fresh_atom L →
      x = (atos a) →
      instantiate As (g_type c cfg gx) = Some (Tss, Tsx) →
      spitoj_type_list As = Ajs →
      spitoj_type Tsx = (TC Tx) →
      spitoj_gproc c cns (a :: L) (x :: vns) (gamma_add g x (RefType (TC Tx)))
                (open_gproc_wrt_term G (term_Var_f a)) J eg →
      lc_gproc H →
      spitoj_gproc c cns L vns g H J eh →
      spitoj_gproc c cns L vns g (:let (dtor_Dtor cfg gx As ts) :in G :else H)
                J (let_expr x Tx gn cfg Ajs es eg eh)
| spitoj_gproc_gen :
    ∀ c cns L vns g G T Tj J a x cfg eg cargs,
      (∀ x, ¬ In x L → lc_gproc (open_gproc_wrt_term G (term_Var_f x))) →
      a = fresh_atom L →
      x = (atos a) →
      ((∃ T', T = type_Channel cfg T'
        ∧ cargs = nil)
       ∨ (∃ n, ∃ Ts', T = type_Nested cfg n Ts'
           ∧ cargs = (ctor_args (ctors_for_type c n (f_idents c))))) →
      spitoj_type T = (TC Tj) →
      spitoj_gproc c cns (a :: L) (x :: vns) (gamma_add g x (RefType (TC Tj)))
                (open_gproc_wrt_term G (term_Var_f a)) J eg →
      spitoj_gproc c cns L vns g (:gen T :in G) J (gen_expr x Tj cfg cargs eg)
| spitoj_gproc_fork :
    ∀ c gcns hcns L vns g G H J eg eh Gn Hn Gvn Hvn Ts,
      lc_gproc G →
      lc_gproc H →
      length Ts = length vns →
      (∀ vn T, In vn vns → g vn = Some T → In T Ts) →
      J Gn = Some (new_thread Gn vns Ts eg) →
      J Hn = Some (new_thread Hn vns Ts eh) →
      spitoj_gproc c gcns L vns g G J eg →
      spitoj_gproc c hcns L vns g H J eh →
      Gn = fresh_cname (gcns ++ hcns) →
      Hn = fresh_cname (Gn :: gcns ++ hcns) →
      Gvn = fresh_vname vns →
      Hvn = fresh_vname (Gvn :: vns) →
      spitoj_gproc c (Hn :: Gn :: gcns ++ hcns) L vns g (G :| H) J
                (fork_expr Gn Hn Gvn Hvn (map Var vns))
| spitoj_gproc_null :
    ∀ c cns L vns g J,
      spitoj_gproc c cns L vns g (gproc_null) J (NOP).
```

```
End ParametrizedGlobalPi.
```

Listing 51: Transforming Parallel Composition, Definition of `spitoj_gproc`

4.3.6. The Final Result

We combine the transformations of the various parts of the Global Expi calculus discussed above in the main transformation relation `spitoj`. We transform the configuration, a list of fresh class names, a Global Expi process and a list of free Expi names that occur in that process into a VPJ program (i.e., class declarations) and a target expression. The resulting program contains declarations of the standard VPJ classes (i.e., `Object`, `Thread` etc.) defined by the relation `has_standard_classes` (see Appendix A.4), the symbolic library classes, as defined by `has_symbolic_library` (see Appendix A.10) and the custom classes generated by the relation `spitoj_type_class`. Furthermore, we declare the class `Fun` and add transformations of all constructors and destructors to it. Finally, we transform the names of free Expi names occurring in the Global Expi process to local variable names, declare them using the function `main_expr` and transform the Global Expi process in the scope of these local variables.

```
Section ParametrizedGlobalPi.

Fixpoint  main_expr (vnts : list (vname × jtype)) (e : expr) : expr :=
match vnts with
 | nil ⇒
     e ; NOP
 | (v, RefType (TC T)) :: xs ⇒
     Block v (RefType (TC T)) None (v ::= jnull ; (main_expr xs e))
 | (v, T) :: xs ⇒
     Block v T (Some (default_val T)) (main_expr xs e)
end.

Inductive  spitoj : @config Idents →
                    list cname →
                    list (atom × @type Idents) →
                    @gproc Idents →
                    jprog →
                    expr →
                    Prop :=
 |  spitoj_rule :
     ∀ c cns fnts G J e vnts main fun_fds fun_mds,
       (* standard classes *)
       has_standard_classes J →
       (* symbolic library *)
       has_symbolic_library J →
       (* custom types *)
       NoDup cns →
       (∀ tn cn,
        In tn (t_idents c) →
        cn = idtostr tn →
        In cn cns →
```

```
      spitoj_type_class c tn cn J) →
  (* constructor/destructor implementation class *)
  In Fun cns →
  J Fun = Some (nil, CObject, fun_fds, fun_mds) →
  (* constructors *)
  (∀ fn,
   In fn (f_idents c) →
     fun_mds (ctor_mname (idtostr fn)) = (ctor_method c fn)) →
  (* destructors *)
  (∀ gn,
   In gn (g_idents c) →
     fun_mds (dtor_mname (idtostr gn)) = (dtor_method c gn)) →
  (* free names *)
  NoDup fnts →
  vnts = (map (fun p ⇒ (atos (fst p), RefType (spitoj_type (snd p)))) fnts) →
  (* main process *)
  main = (main_expr vnts e) →
  lc_gproc G →
  spitoj_gproc c cns (map fst fnts) (map fst vnts) (gamma_init vnts) G J e →
  spitoj c cns fnts G J main.

End ParametrizedGlobalPi.
```

Listing 52: Remaining Transformation Definitions

4.3.7. Implementation Notes

The implementation of the transformation from Expi Calculus to Variant Parametric Jinja differs in some details from the model we present in this thesis. Most of these differences arise from the simplifications we had to make in order to obtain a clean model and can be seen as syntactic sugar. For example, the type annotations in Expi terms are optional in the implementation and can be automatically inferred in most cases. We also allow to omit the `proc_null` process when it is used as a continuation process to make the processes easier to read.

VPJ, on the other hand, has some small semantic differences to Java. Unlike in Java, there are no statements to control the execution flow of a VPJ expression, such as **break**, **continue** or **return**. While controlling the execution flow in the loops is used not so often, the **return** is a very common statement in Java. In VPJ, a method returns its whole body as one big expression that is reduced to something before it is used. Therefore, in order to return a value one needs to make sure that the body reduces to a `Val` expression. This also has an implication on the conditional expression `Cond`. The version used in VPJ is similar to the Java's ternary operator "_ ? _ : _", it always needs both branches to be present and to have the same type. This makes it impossible to return a value in one of the branches and throw an exception in another, because the type of a `throw` statement is the thrown exception. One has to return a dummy value of the right type just after the `throw` to make the code pass the type checker.

4.4. Proofs

We have proved that the symbolic library and the VPJ code generated by the transformation is well-typed. This is an important consistency check that shows that our definitions are at least not completely broken. For example, it helped us to find that the first version of our transformation was using the field access in an inconsistent way. We have also found and fixed some other minor problems, like the difference in the typing of a **throw** statement in Java and VPJ. In the longer-term perspective these proofs will be needed to prove the correctness of our transformation.

The proof that the symbolic library is well-typed shows that each class of the symbolic library is well-typed (as defined by `is_wt_class`) in a VPJ program containing standard class declarations and the symbolic library classes. To be more precise, we show the following lemmas:

```
Lemma  wt_AbstractBase : ∀ J (Hst : has_standard_classes J)
                           (H : has_symbolic_library J),
                              is_wt_class J "AbstractBase".

Lemma  wt_AbstractGenerativeBase : ∀ J (Hst : has_standard_classes J)
                                     (H : has_symbolic_library J),
                                        is_wt_class J "AbstractGenerativeBase".

Lemma  wt_ELibrary : ∀ J (Hst : has_standard_classes J) (H : has_symbolic_library J),
                       is_wt_class J "ELibrary".

Lemma  wt_ELetProcess : ∀ J (Hst : has_standard_classes J)
                          (H : has_symbolic_library J),
                             is_wt_class J "ELetProcess".

Lemma  wt_EDestructor : ∀ J (Hst : has_standard_classes J)
                          (H : has_symbolic_library J),
                             is_wt_class J "EDestructor".

Lemma  wt_Semaphore : ∀ J (Hst : has_standard_classes J) (H : has_symbolic_library J),
                        is_wt_class J "Semaphore".

Lemma  wt_AbstractChannel : ∀ J (Hst : has_standard_classes J)
                              (H : has_symbolic_library J)
                              (Hwf : well_founded (superclass1 J)),
                                 is_wt_class J "AbstractChannel".
```

Listing 53: Typing Symbolic Library

We have shown that all used types are well-formed and all declared methods are well-typed. Most of the symbolic classes have only a simple constructor method that initializes the fields. The classes `Semaphore` and `AbstractChannel` also have more complicated methods. The `Semaphore` has methods `acquire()` and `release()` (see Appendix A.7.3) and the `AbstractChannel` has methods `send(T)` and `receive()` (see Appendix A.7.4).

The proof is in all cases by case analysis on the corresponding expression and using the right case of the expression typing relation. We need to give the correct type of each subexpression and show the premises of the typing rules. These proofs are not complicated, but quite long and tedious.

Showing that the code generated by the transformation is well-typed is much harder. Most typing rules require providing the exact types of all subexpressions and only fail in the last moment if several (wrong) alternative types are possible. For example, typing field assignment (see `wte_fass` in Appendix A.4) can be quite tricky. Another problem was finding the right invariants to type check the code generated by recursive functions. An incorrect invariant (i.e., the type of the expression we are trying to type check and the preconditions) usually becomes noticeable only when applying the induction hypothesis in the proof.

We prove that the expressions generated for the Global Expi processes, the constructor and destructor methods and the classes representing Expi types are well-typed assuming that our invariants hold. In the end, we can use these results to show the following theorem:

```
Theorem  translation_is_wt : ∀ c cns L G J h e
                               (Hnd1 : NoDup cns)
                               (Hnd2 : NoDup L)
                               (Hlc : lc_gproc G)
                               (Hh : is_preallocated h)
                               (Hwf : well_founded (superclass1 J))
                               (H : spitoj atos Idents idtostr c cns L G J e),
                                  is_wf_prog J (h, stack_empty) e.
```

Listing 54: Typing Generated Code

Unfortunately, we could not prove all lemmas in full detail due to the lack of time. First of all, we have not proved that the transformation function preserves the invariants we use to prove the generated VPJ code well-typed. Furthermore, we omitted proving complicated list rewriting lemmas, some substitution rewriting lemmas and other similar helper lemmas in cases that looked trivially true but were tedious to prove in Coq. We have also assumed that the transformation of Expi terms and the declaration of free Expi names are well-typed.

5. Implementation

Our transformation is implemented in a code generation tool named expi2java. The first version of this tool was presented in the author's bachelor's thesis [Bus08] and was very much improved since then. In this chapter we will briefly describe the features of expi2java and give an overview over the most important changes we made since the first version.

5.1. Expi2java

Expi2java[1] is a code generator for security protocols designed to be highly customizable and extensible. It takes protocol specifications written in the Extensible Spi Calculus, configurations that provide the low-level information needed for code generation and produces interoperable Java code. Expi2java is free software, it is distributed under the terms of GPLv3.

The syntax of Expi calculus we use in expi2java is similar to the one used in the protocol verifier ProVerif [Bla08] and includes limited support for some ProVerif-specific language constructs such as events and queries. The main difference between our syntax and the variant of applied pi-calculus used in ProVerif is that our calculus is typed and we therefore need to annotate some terms with their types. Expi2java can pretty-print the protocol specifications in ProVerif syntax which allows the user to verify the protocol model before generating the implementation.

The user can customize and extend the input language by defining new types, cryptographic primitives and configurations. The configurations play a bigger role in the implementation than in the model of Expi calculus presented in Chapter 2. Besides the custom types and cryptographic primitives, expi2java also provides a way to specify what class should be used to implement the corresponding cryptographic primitive and allows to pass user-defined parameters to the implementation class. This can be used for example to specify which encryption mode, padding and algorithm should be used to encrypt data or to specify the length and endianness of an integer to match the protocol specification, basically giving the user full control over the low-level data format in the generated protocol messages.

In addition to the symbolic library presented in Section 3.8, we also provide a concrete library that implements real networking and cryptography. The concrete library contains implementation of the most common cryptographic primitives and data types out of the box, but it can also easily be extended by the user. We use the standard Java cryptographic providers to ensure interoperability with other existing protocol implementations.

[1] Home page: http://www.infsec.cs.uni-saarland.de/projects/expi2java

The code generation phase is also customizable. Expi2java uses special snippets of Java code to generate the protocol implementation. The user can customize the snippets and the class stubs to simplify integration of the generated code into existing applications. The transformation presented in Chapter 4 models the version of the default transformation used in the tool, with only minor simplifications.

5.2. TLS Case Study

In order to show the potential of expi2java we have generated a fully functional and interoperable implementation of the TLS protocol (Transport Layer Security). The first version of the TLS model presented in the author's bachelor's thesis only had support for the client side of the protocol and one encryption scheme. Recently, we have extended the model with the server side, dynamic encryption scheme selection and support for proper error handling.

The current version of the TLS model implements TLS v1.0 [DA99] with the AES extension [Cho02] and the Server Name Indication (SNI) extension [BWNH+06]. The model includes both client and server sides, the handshake, the application data protocol and the alert protocol used for error reporting. We support 6 different encryption schemes (including AES, RC4 and 3DES algorithms with different key lengths, SHA1 or MD5 HMAC and RSA key exchange). One of these encryption schemes is dynamically chosen during the handshake. The model consists of an Expi process (about 850 lines) and a configuration file (625 lines). The transformation to Java only takes about 12 seconds on a laptop with a Core2 Duo P7450 CPU. We have tested the generated implementation for interoperability with the common browsers and web servers.

We have verified some security properties (secrecy of the nonces and the private key) of our TLS model with ProVerif. The verification process for 14 queries took about 5 minutes. Unfortunately, we have experienced termination problems with more complicated queries.

5.3. Recent Improvements

In the 5 releases of expi2java we made a lot of progress in form of both practical features and usability improvements, taking expi2java from a prototype into a mature and useful tool. One of the most interesting new features is the type inference for the type annotations of Expi constructor and destructor applications. The type inference significantly simplifies specifications of large protocols and improves readability of the protocol models.

Another important new feature is the support for parameterizing the processes with configurations. This can be used to simplify models of complex and flexible protocols that can dynamically select between several similar settings, such as different encryption schemes in TLS. Using the parameterized processes, we could add support for 6 different encryption schemes to our old model of TLS (which was supporting only one encryption scheme) with only a few lines.

We have added a couple of new implementation classes to the concrete library. For example, we added a primitive for a TLS channel generated from a protocol specification verified with ProVerif. We have also added more example protocols and extended the set of default configurations.

The addition of the symbolic library for cryptography and networking is another of the recent improvements we have made to expi2java. We keep the interface of both the symbolic and the concrete library equal, so that they can be used interchangeably. The symbolic library is not only useful for specifying security properties in an abstract way, but can also be very useful in practice for debugging purposes when writing a protocol model.

Finally, we have substantially improved the documentation of the tool. We wrote a detailed user manual[2] that gives an overview over the design of expi2java, describes the input language and explains how to extend the input language, the concrete library and how to customize the code generation phase. Additionally, we have created a detailed tutorial[3] that explains how to design a model for a cryptographic protocol, write it in Expi calculus and generate a working implementation with expi2java.

[2] http://www.infsec.cs.uni-saarland.de/projects/expi2java-releases/expi2java-manual-1.5.pdf
[3] http://www.infsec.cs.uni-saarland.de/projects/expi2java-releases/tutorial-1.5.pdf

6. Related Work

We are aware of several projects aimed at generating protocol implementations from abstract models. Most code generators, such as the AGVI toolkit by A. Perrig, D. Song and D. Phan [PSP01], CIL2Java by J. Millen and F. Muller [MM01], SPEAR II by S. Lukell, C. Veldman and A. Hutchison [LVH03] and the Sprite tool by B. Tobler [Tob05] are experimental tools that cannot be used in practice to generate implementations of real-life protocols. Only the spi2java framework by A. Pironti et al. [Pir10] is advanced enough to be able to generate interoperable implementations of relatively complex protocols such as SSH [PS07]. Nevertheless, the practicality of this tool is quite limited; it is hard to work with and is not flexible enough to support really complex protocols like TLS. Finally, none of the projects we are aware of have a thorough formalization of their transformation and have not proved preservation of any security properties.

We based our formalization of the target language on Jinja with Threads by A. Lochbihler [Loc08], the most comprehensive subset of Java we could find. Jinja with Threads is in turn based on Jinja, a single-threaded subset of Java developed by G. Klein and T. Nipkow [KN06]. We have also considered other fragments, such as Bicolano MT, a multi-threaded subset of Java by M. Huisman and G. Petri [HP08]. This work is focused on the formalization of Java semantics at byte code level and does not fit our needs very well. Other subsets of Java, such as the Generic Featherweight Java by Igarashi et al. [IPW01] and Lightweight Java by R. Strniša, P. Sewell and M. Parkinson [SSP07] are too limited for our purposes and were not formalized before.

Interesting alternatives to Java as the target language would be Scala[1] and F#[2]. The semantics of these languages is closer to the Expi calculus, they support immutable data structures, structural equality and pattern matching out of the box, which would considerably simplify our translation. On the other hand, we are not aware of any mechanized formalizations of (comprehensive subsets of) these languages and spending a lot of time and effort just to formalize the target language was clearly not an option.

Another related research field are verifying compilers. Although the focus in this field does not lie on implementation of cryptographic protocols, the problem that needs to be solved is quite similar, one needs to show that the transformation from one language to another is correct and preserves some properties of interest. Some interesting examples of verified compilers are CompCert by X. Leroy [Ler09] and Concurrent C-Minor by A. Appel et al. [App11].

[1] http://www.scala-lang.org/
[2] http://www.fsharp.net/

7. Conclusion

7.1. Results

Formalization We have defined a mechanized formalization of the Extensible Spi Calculus in the Coq proof assistant. The Expi calculus is an extensible variant of the Spi calculus [AB05] with configurations and a type system with parametric types, defined in author's bachelor's thesis. The main challenge in the formalization was finding a way to keep the definitions flexible and extensible enough to represent user-defined types, Expi constructors and destructors in Coq. We also have defined the default configuration containing some common cryptographic primitives and data types and proved its consistency.

Furthermore, we have formalized Variant Parametric Jinja based on a subset of Java called "Jinja with Threads" [Loc08]. We have translated the formalization of Jinja with Threads from Isabelle/HOL to Coq and extended it with the type system from Variant Parametric Featherweight Java [IV06]. The resulting language supports concurrency with shared memory and synchronization, exception handling and a sophisticated type system with parametric polymorphism and subtyping.

Our main contribution is the formalization of the transformation from Expi to VPJ. We have defined an intermediate language, the Global Expi calculus based on the idea of an alternative "global" semantics for the pi-calculus [Wis04]. The transformation happens in two steps, first we translate the protocol specification from the Expi calculus to the Global Expi calculus, then we transform the protocol model in Global Expi calculus into a VPJ program. The generated VPJ program uses a symbolic library that represents cryptographic operations as abstracts terms and implements pi-calculus channels by communication between local threads. The transformation of Expi destructors implements structural equality and pattern matching on terms.

Module	Lines of code
Generated LNgen lemmas	12981
Formalization of Expi and Global Expi calculi	953
Expi proofs	2864
Formalization of VPJ	6979
Symbolic library	510
Transformation from Global Expi to VPJ	1066
VPJ proofs (symbolic library and the transformation are well-typed)	3679
Auxiliary definitions and lemmas	1217
Total sum (without LNgen lemmas)	17268

Table 7.1.: Formalization Size

Our formalization consists of several modules. We use Ott and LNgen to formalize both Expi and Global Expi calculi; the resulting formalization of these two calculi is only 953 lines long. Additionally, LNgen generate a large collection of various useful lemmas, this "metatheory library" is 12981 lines long. The definition of the default configuration and the corresponding proofs take 2864 lines of Coq code. The formalization of VPJ is much larger, it takes 6979 lines of code. The transformation consists of 510 lines of symbolic library declarations and 1066 lines for the transformation from Global Expi to VPJ. The proof that the symbolic library and the code generated by our transformation is well-typed took 3679 lines of code. In total (not counting the generated metatheory library) our formalization requires 17268 lines of Coq code.

Proved Transformation Well-Typed We have proved that the symbolic library and the VPJ program generated by the transformation of a well-typed Expi process is well-typed. We plan to use this proof in the future to show the correctness of our transformation. Even now, having a well-typed program already significantly increases the confidence in the transformation. It shows that the transformation of the variant parametric types is correct.

Improved Implementation Finally, we also have significantly improved expi2java, our tool that implements the transformation from Expi to VPJ. Expi2java has grown from a prototype into a mature and useful tool. It is well documented and is flexible enough to generate interoperable protocol implementations for very complex protocols. We have extended our model of TLS with a server part and dynamic encryption scheme selection, verified some security properties of the TLS model with ProVerif and generated an interoperable implementation. The implementation of the TLS server was shown to work with popular browsers like Firefox and Opera.

7.2. Future Work

Show that the Transformation Preserves Trace Properties One intresting direction for future work would be to show that our transformation preserves all trace properties of the original protocol. An interesting example of trace properties are the safety properties, i.e., sets of traces closed under taking prefixes. One way to define safety properties are correspondence assertions [WL93]. Model-checkers can check safety properties by examining all traces produced by the reduction of the protocol model (usually up to a bounded depth, for a bounded number of participants, etc.). The events represent observations of a passive attacker (for example messages sent over public channels) or mark important stages reached in a protocol run (such as the start and the end of an authentication phase).

When a protocol satisfies a trace property, it means that the set of all traces that can be produced by a protocol run is a subset of the trace property in question. If we can prove that the set of traces produced by the transformation of a protocol is a subset of the traces of the original protocol model, it would imply that the generated protocol implementation satisfies all trace properties satisfied by the protocol model. As a consequence, the protocol

implementation generated from a safe protocol model (i.e., one that satisfies some interesting safety properties) would also be safe.

Proving that our transformation preserves trace properties poses several challenges. First of all, we need to represent the protocol events on the VPJ side and to define the transformation of Expi events to VPJ. Furthermore, we need to relate very different representations of traces to show trace inclusion. An interesting proof technique that could simplify this task is the so called (bi)simulation. In order to use simulation we would need to define a labeled transition system on each sides of the transformation with protocol events as the labels. We could then use weak labeled simulation to relate VPJ programs to Expi processes. Showing only one direction of simulation would be sufficient for our purposes, since it would already imply trace inclusion and therefore the desired preservation of trace properties.

Show that the Transformation Preserves Security (Robust Safety) Even more interesting would be to show that the transformation preserves the security of the original protocol. For this we could use a stronger notion of safety usually called *robust safety*, i.e., safety for all attackers. We can define the set of robust traces of a protocol as the union of all traces that can be produced by this protocol when run in the presence of an arbitrary attacker. A protocol is robustly safe with respect to some safety property if the set of its robust traces is a subset of that safety property. For example, ProVerif [Bla01] can check the robust safety of protocols with respect to (weak) secrecy and authenticity properties.

In the Expi calculus, the presence of an attacker is modeled by a contextual Expi process that does not raise any events. In VPJ we would need to model the attacker as an arbitrary VPJ expression running in a concurrent thread. Therefore, directly showing the inclusion of robust traces produced by the transformation into the set of robust traces of the original protocol model would require defining a backward transformation from VPJ to Expi, because we would need to relate an arbitrary VPJ attacker to an Expi attacker.

Instead of defining this backward transformation, we could prove a slightly weaker notion of robust safety that relates the Expi process in the presence of all Expi attackers with a VPJ program executed in the presence of the *transformations* of all the Expi attackers. This weaker property can probably be shown by defining a weak labeled (bi)simulation as discussed above, and then proving that this simulation is contextual with respect to Expi contexts. Proving the simulation for our transformation in the presence of a transformation of an arbitrary Expi attacker would imply robust trace inclusion and therefore preservation of robust safety with respect to Expi calculus contexts.

Show Functional Correctness of the Transformation Another important property of the transformation is the functional correctness. Intuitively, a transformation is correct if the resulting program preserves the intended message flow of the original protocol. It is hard to relate messages sent in such different languages as Expi calculus and VPJ, so the obvious solution would be to use a similar technique as for proving the preservation of safety properties. The difference to the proof of preservation of safety properties is that in this case we need to show the other direction of the set inclusion, i.e., that the traces produced by the Expi process are a subset of the corresponding traces produced by the transformation of that process.

Bibliography

[AB01] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Proc. 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2030 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, 2001. 5

[AB05] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005. v, 9, 11, 17, 87

[Aba99] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999. 5

[ACP+08] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proc. 35th Symposium on Principles of Programming Languages (POPL '08)*, pages 3–15, 2008. 10, 26, 31

[AG99] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998. 5

[Ahm04] Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004. 31

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23:657–683, September 2001. 31

[App11] Andrew W. Appel. Verified software toolchain (invited talk). In Gilles Barthe, editor, *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011. 85

[AW10] Brian E. Aydemir and Stephanie Weirich. LNgen: Tool support for locally nameless representations. Draft available at http://www.cis.upenn.edu/~sweirich/papers/lngen/, 2010. 31

[BCFM07] M. Backes, A. Cortesi, R. Focardi, and M. Maffei. A calculus of challenges and responses. In *Proc. 5th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 101–116. ACM Press, 2007. 5

[BFM07] Michele Bugliesi, Riccardo Focardi, and Matteo Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563–617, 2007. 5

[BGHM09] Michael Backes, Martin P. Grochulla, Cătălin Hriţcu, and Matteo Maffei. Achieving security despite compromise using zero-knowledge. In *22th IEEE Symposium on Computer Security Foundations (CSF 2009)*, pages 308–323. IEEE Computer Society Press, July 2009. 11

[BHM08] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Type-checking zero-knowledge. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 357–370. ACM Press, October 2008. 5, 11

[Bla01] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE Computer Society Press, 2001. 5, 89

[Bla08] Bruno Blanchet. *ProVerif v1.14pl4 (Automatic Cryptographic Protocol Verifier) User Manual*. CNRS, Département d'Informatique, École Normale Supérieure, Paris, February 2008. http://www.proverif.ens.fr/. 81

[BTPF08] Christoph Benzmüller, Frank Theiss, Larry Paulson, and Arnaud Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. In *4th International Joint Conference on Automated Reasoning, IJCAR 2008*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008. 31

[Bus08] Alex Busenius. Expi2Java – An extensible code generator for security protocols. Bachelor's Thesis, 2008. 5, 6, 15, 20, 81

[BWNH+06] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. *Transport Layer Security (TLS) Extensions*, April 2006. RFC 4366. 82

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, 1992. 20

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76:95–120, February 1988. 34

[Cho02] P. Chown. *Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)*, June 2002. RFC 3268 (Informational). 82

[CPM90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In *Proceedings of the international conference on Computer logic*, pages 50–66, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 34

[DA99] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*, January 1999. RFC 2246 (Proposed Standard). 7, 82

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381 – 392, 1972. 10

[FGM07] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF)*, pages 31–45. IEEE Computer Society Press, 2007. 5, 11

[GJ04] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3):435–484, 2004. 5

[Gor93] Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *6th International Workshop on Higher-order Logic Theorem Proving and its Applications (HUG '93)*, pages 413–425, 1993. 10

[HJ06] Christian Haack and Alan Jeffrey. Pattern-matching spi-calculus. *Information and Computation*, 204(8):1195–1263, 2006. 5

[HP08] Marieke Huisman and Gustavo Petri. BicolanoMT: a formalization of multi-threaded Java at bytecode level. In *Bytecode 2008*, Electronic Notes in Theoretical Computer Science, 2008. 85

[HS09] Cătălin Hriţcu and Jan Schwinghammer. A step-indexed semantics of imperative objects. *Logical Methods in Computer Science (LMCS)*, 5(4:2):1–48, December 2009. 31

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, Mai 2001. 34, 85

[IV06] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, 28:795–847, September 2006. v, 6, 33, 34, 45, 46, 49, 87

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):619–695, 2006. 33, 43, 85

[Ler09] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43:363–446, December 2009. 85

[Loc08] Andreas Lochbihler. Type safe nondeterminism - a formal semantics of java threads. In *International Workshop on Foundations of Object-Oriented Languages (FOOL 2008)*, January 2008. v, 6, 33, 43, 52, 85, 87

[Loc10] Andreas Lochbihler. Verifying a compiler for java threads. In A. D. Gordon, editor, *European Symposium on Programming (ESOP'10)*, pages 427–447. Springer, March 2010. 55

[LVH03] Simon Lukell, Christopher Veldman, and Andrew Hutchison. Automated attack analysis and code generation in a multi-dimensional security protocol engineering framework. In *Southern African Telecommunications Networks and Applications Conference*. University of Cape Town, 2003. 85

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. 39

[MM01] J. Millen and F. Muller. Cryptographic protocol generation from CAPSL. Technical Report SRI-CSL-01-07, SRI International, December 2001. 5, 85

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT press, 2002. 20, 21

[Pir10] Alfredo Pironti. *Sound Automatic Implementation Generation and Monitoring of Security Protocol Implementations from Verified Formal Specifications*. PhD thesis, Politecnico di Torino (Italy), 2010. 85

[PNW11] Larry Paulson, Tobias Nipkow, and Makarius Wenzel. Isabelle proof assistant, 2011. Version 2011, Home page: http://www.cl.cam.ac.uk/research/hvg/Isabelle/. 33

[PPM90] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Proceedings of the fifth international conference on Mathematical foundations of programming semantics*, pages 209–226, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 34

[PS07] Alfredo Pironti and Riccardo Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *International Symposium on Computers and Communications (ISCC)*, January 2007. 5, 85

[PSP01] A. Perrig, D. Song, and D. Phan. AGVI – Automatic Generation, Verification, and Implementation of security protocols. In *Proc. Computer Aided Verification'01 (CAV)*, Lecture Notes in Computer Science. Springer-Verlag, 2001. 85

[SNO+10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *The Journal of Functional Programming*, 20(1):71–122, 2010. Home page: http://www.cl.cam.ac.uk/~pes20/ott/. 31

[SSP07] Rok Strniša, Peter Sewell, and Matthew Parkinson. The java module system: core design and semantic definition. *SIGPLAN Not.*, 42:499–514, October 2007. 85

[Tea10] The Coq Development Team. The Coq proof assistant, 2010. Version 8.3, Home page: http://coq.inria.fr/. v, 6, 9

[THE+04] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal M. Gafter. Adding wildcards to the java programming language. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC)*, pages 1289–1296, 2004. 34

[Tob05] Benjamin Tobler. A structured approach to network security protocol implementation. Master's thesis, University of Cape Town, November 2005. 5, 85

[Wis04] Lucian Wischik. Old names for nu, 2004. Presented at Dagstuhl Seminar 04241. 59, 62, 87

[WL93] Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, SP '93, pages 178–, Washington, DC, USA, 1993. IEEE Computer Society. 88

A. Appendix

A.1. Expi Calculus: Library Functions

```
Section ParametrizedByName.
```

A.1.1. Auxiliary Definitions

```
Fixpoint  list_assoc (A B : Set) (eq : ∀ a b : A, {a = b} + {a ≠ b}) (x : A)
                  (l : list (A × B)) {struct l} : option B :=
match l with
 | nil ⇒ None
 | cons (a,b) t ⇒ if (eq a x) then Some b else list_assoc A B eq x t
end.

Fixpoint  open_term_wrt_term_rec (k : nat) (t5 : term) (t_6 : term) : term :=
match t_6 with
 | term_Nam nam5 ⇒ term_Nam nam5
 | term_Var_b nat ⇒
    match lt_eq_lt_dec nat k with
     | inleft (left _) ⇒ term_Var_b nat
     | inleft (right _) ⇒ t5
     | inright _ ⇒ term_Var_b (nat - 1)
    end
 | term_Var_f x ⇒ term_Var_f x
 | term_Ctor cfg n tl xl ⇒
    term_Ctor cfg n tl (List.map (open_term_wrt_term_rec k t5) xl)
end.
Definition  open_term_list_wrt_term_rec (k : nat) (t : term) (xl : list term)
                                       : list term :=
  List.map (open_term_wrt_term_rec k t) xl.

Fixpoint  open_nam_wrt_nam_rec (k : nat) (a : nam) (b : nam) : nam :=
match b with
 | Nam_b n ⇒ match lt_eq_lt_dec n k with
              | inleft (left _) ⇒ Nam_b n
              | inleft (right _) ⇒ a
              | inright _ ⇒ Nam_b (n - 1)
             end
 | Nam_f s ⇒ Nam_f s
end.

Definition  open_nam_wrt_nam b a := open_nam_wrt_nam_rec 0 a b.
```

```
Fixpoint  open_term_wrt_nam_rec (k : nat) (a : nam) (t : term) : term :=
match t with
 | term_Nam nn ⇒ term_Nam (open_nam_wrt_nam_rec k a nn)
 | term_Var_b n ⇒ term_Var_b n
 | term_Var_f x ⇒ term_Var_f x
 | term_Ctor cfg n tl xl ⇒ term_Ctor cfg n tl (List.map (open_term_wrt_nam_rec k a) xl)
end.
Definition  open_term_list_wrt_nam_rec (k : nat) (n : nam) (xl : list term) : list term :=
  List.map (open_term_wrt_nam_rec k n) xl.

Definition  open_dtor_wrt_term_rec (k : nat) (t5 : term) (g5 : dtor) : dtor :=
match g5 with
 | dtor_Dtor cfg n tl xl ⇒ dtor_Dtor cfg n tl (open_term_list_wrt_term_rec k t5 xl)
end.

Definition  open_dtor_wrt_nam_rec (k : nat) (nam5 : nam) (g5 : dtor) : dtor :=
match g5 with
 | dtor_Dtor cfg n tl xl ⇒ dtor_Dtor cfg n tl (open_term_list_wrt_nam_rec k nam5 xl)
end.

Fixpoint  open_proc_wrt_term_rec (k : nat) (t5 : term) (P5 : proc) {struct P5}:  proc :=
match P5 with
 | proc_out t u P ⇒ proc_out (open_term_wrt_term_rec k t5 t)
                             (open_term_wrt_term_rec k t5 u)
                             (open_proc_wrt_term_rec k t5 P)
 | proc_in t P ⇒ proc_in (open_term_wrt_term_rec k t5 t)
                         (open_proc_wrt_term_rec (S k) t5 P)
 | proc_bangin t P ⇒ proc_bangin (open_term_wrt_term_rec k t5 t)
                                 (open_proc_wrt_term_rec (S k) t5 P)
 | proc_let g P Q ⇒ proc_let (open_dtor_wrt_term_rec k t5 g)
                             (open_proc_wrt_term_rec (S k) t5 P)
                             (open_proc_wrt_term_rec k t5 Q)
 | proc_new T P ⇒ proc_new T (open_proc_wrt_term_rec k t5 P)
 | proc_fork P Q ⇒ proc_fork (open_proc_wrt_term_rec k t5 P)
                             (open_proc_wrt_term_rec k t5 Q)
 | proc_null ⇒ proc_null
end.

Fixpoint  open_proc_wrt_nam_rec (k : nat) (nam5 : nam) (P5 : proc) {struct P5}:  proc :=
match P5 with
 | proc_out t u P ⇒ proc_out (open_term_wrt_nam_rec k nam5 t)
                             (open_term_wrt_nam_rec k nam5 u)
                             (open_proc_wrt_nam_rec k nam5 P)
 | proc_in t P ⇒ proc_in (open_term_wrt_nam_rec k nam5 t)
                         (open_proc_wrt_nam_rec k nam5 P)
 | proc_bangin t P ⇒ proc_bangin (open_term_wrt_nam_rec k nam5 t)
                                 (open_proc_wrt_nam_rec k nam5 P)
 | proc_let g P Q ⇒ proc_let (open_dtor_wrt_nam_rec k nam5 g)
                             (open_proc_wrt_nam_rec k nam5 P)
                             (open_proc_wrt_nam_rec k nam5 Q)
 | proc_new T P ⇒ proc_new T (open_proc_wrt_nam_rec (S k) nam5 P)
 | proc_fork P Q ⇒ proc_fork (open_proc_wrt_nam_rec k nam5 P)
                             (open_proc_wrt_nam_rec k nam5 Q)
```

```
 | proc_null ⇒ proc_null
end.

Definition  open_term_wrt_term t5 t_6 := open_term_wrt_term_rec 0 t_6 t5.

Definition  open_term_wrt_nam nam_6 t5 := open_term_wrt_nam_rec 0 t5 nam_6.

Definition  open_term_list_wrt_term t5 xl5 := open_term_list_wrt_term_rec 0 xl5 t5.

Definition  open_term_list_wrt_nam nam5 xl5 := open_term_list_wrt_nam_rec 0 xl5 nam5.

Definition  open_dtor_wrt_term t5 g5 := open_dtor_wrt_term_rec 0 g5 t5.

Definition  open_dtor_wrt_nam nam5 g5 := open_dtor_wrt_nam_rec 0 g5 nam5.

Definition  open_proc_wrt_term t5 P5 := open_proc_wrt_term_rec 0 P5 t5.

Definition  open_proc_wrt_nam nam5 P5 := open_proc_wrt_nam_rec 0 P5 nam5.
```

A.1.2. Locally-Closed Terms and Processes

```
Inductive  lc_nam : nam → Prop :=
 |  lc_Nam_f : ∀ (a : atom),
     (lc_nam (Nam_f a)).

Inductive  lc_term_list : list term → Prop :=
 |  lc_term_list_nil :
     (lc_term_list nil )
 |  lc_term_list_cons : ∀ (t : term) (xl : list term),
     (lc_term t) →
     (lc_term_list xl) →
     (lc_term_list ( t :: xl ) )
with  lc_term : term → Prop :=
 |  lc_term_Nam : ∀ (nam5 : nam),
     lc_nam nam5 →
     (lc_term (term_Nam nam5))
 |  lc_term_Var_f : ∀ (x : atom),
     (lc_term (term_Var_f x))
 |  lc_term_Ctor : ∀ (cfg : cfg_name) (n : Idents) (tl : list type) (xl : list term),
     (lc_term_list xl) →
     (lc_term (term_Ctor cfg n tl xl)).

Inductive  lc_dtor : dtor → Prop :=
 |  lc_dtor_Dtor : ∀ (cfg : cfg_name) (n : Idents) (tl : list type) (xl : list term),
     (lc_term_list xl) →
     (lc_dtor (dtor_Dtor cfg n tl xl)).

Inductive  lc_proc : proc → Prop :=
 |  lc_proc_out : ∀ (t u : term) (P : proc),
     (lc_term t) →
     (lc_term u) →
     (lc_proc P) →
     (lc_proc (proc_out t u P))
```

```
| lc_proc_in : ∀ (t : term) (P : proc),
    (lc_term t) →
     ( ∀ x , lc_proc ( open_proc_wrt_term P (term_Var_f x) ) ) →
    (lc_proc (proc_in t P))
| lc_proc_bangin : ∀ (t : term) (P : proc),
    (lc_term t) →
     ( ∀ x , lc_proc ( open_proc_wrt_term P (term_Var_f x) ) ) →
    (lc_proc (proc_bangin t P))
| lc_proc_let : ∀ (g : dtor) (P Q : proc),
    (lc_dtor g) →
     ( ∀ x , lc_proc ( open_proc_wrt_term P (term_Var_f x) ) ) →
    (lc_proc Q) →
    (lc_proc (proc_let g P Q))
| lc_proc_new : ∀ (T : type) (P : proc),
     ( ∀ a , lc_proc ( open_proc_wrt_nam P (Nam_f a) ) ) →
    (lc_proc (proc_new T P))
| lc_proc_fork : ∀ (P Q : proc),
    (lc_proc P) →
    (lc_proc Q) →
    (lc_proc (proc_fork P Q))
| lc_proc_null :
    (lc_proc proc_null).
```

A.1.3. Free Names and Variables

```
Definition  fn_in_nam (nam5 : nam) : vars :=
match nam5 with
 | Nam_b nat ⇒ {}
 | Nam_f a ⇒ {{a}}
end.

Fixpoint  fv_in_term (t5 : term) : vars :=
match t5 with
 | term_Nam nam5 ⇒ {}
 | term_Var_b nat ⇒ {}
 | term_Var_f x ⇒ {{x}}
 | term_Ctor cfg n tl xl ⇒ (List.fold_left (fun v x ⇒ union (fv_in_term x) v) xl {})
end.
Definition  fv_in_term_list (xl5 : list term) : vars :=
  List.fold_left (fun v x ⇒ union (fv_in_term x) v) xl5 {}.

Fixpoint  fn_in_term (t5 : term) : vars :=
match t5 with
 | term_Nam nam5 ⇒ (fn_in_nam nam5)
 | term_Var_b nat ⇒ {}
 | term_Var_f x ⇒ {}
 | term_Ctor cfg n tl xl ⇒ (List.fold_left (fun v x ⇒ union (fn_in_term x) v) xl {})
end.
Definition  fn_in_term_list (xl5 : list term) : vars :=
  List.fold_left (fun v x ⇒ union (fn_in_term x) v) xl5 {}.

Definition  fv_in_dtor (g5 : dtor) : vars :=
```

```
match g5 with
 | dtor_Dtor cfg n tl xl ⇒ (fv_in_term_list xl)
end.

Definition  fn_in_dtor (g5 : dtor) : vars :=
match g5 with
 | dtor_Dtor cfg n tl xl ⇒ (fn_in_term_list xl)
end.

Fixpoint  fv_in_proc (P5 : proc) : vars :=
match P5 with
 | proc_out t u P ⇒ (fv_in_term t) u (fv_in_term u) u (fv_in_proc P)
 | proc_in t P ⇒ (fv_in_term t) u (fv_in_proc P)
 | proc_bangin t P ⇒ (fv_in_term t) u (fv_in_proc P)
 | proc_let g P Q ⇒ (fv_in_dtor g) u (fv_in_proc P) u (fv_in_proc Q)
 | proc_new T P ⇒ (fv_in_proc P)
 | proc_fork P Q ⇒ (fv_in_proc P) u (fv_in_proc Q)
 | proc_null ⇒ {}
end.

Fixpoint  fn_in_proc (P5 : proc) : vars :=
match P5 with
 | proc_out t u P ⇒ (fn_in_term t) u (fn_in_term u) u (fn_in_proc P)
 | proc_in t P ⇒ (fn_in_term t) u (fn_in_proc P)
 | proc_bangin t P ⇒ (fn_in_term t) u (fn_in_proc P)
 | proc_let g P Q ⇒ (fn_in_dtor g) u (fn_in_proc P) u (fn_in_proc Q)
 | proc_new T P ⇒ (fn_in_proc P)
 | proc_fork P Q ⇒ (fn_in_proc P) u (fn_in_proc Q)
 | proc_null ⇒ {}
end.
```

A.1.4. Substitutions

```
Definition  subst_nam_in_nam (nam5 : nam) (a5 : atom) (nam_6 : nam) : nam :=
match nam_6 with
 | Nam_b nat ⇒ Nam_b nat
 | Nam_f a ⇒ (if eq_var a a5 then nam5 else (Nam_f a))
end.

Fixpoint  subst_type_var_in_type (sub : list (string × type)) (T_6 : type)
                                 {struct T_6} : type :=
match T_6 with
 | type_Top ⇒ type_Top
 | type_Var s5 ⇒ match list_assoc string_dec s5 sub with
                  | None ⇒ type_Var s5
                  | Some T5 ⇒ T5
                 end
 | type_Channel cfg T ⇒ type_Channel cfg (subst_type_var_in_type sub T)
 | type_Nested cfg n tl ⇒ type_Nested cfg n (List.map (subst_type_var_in_type sub ) tl)
end.
Definition  subst_type_var_in_type_list (sub : list (string × type))
```

```
                                    (tl5 : list type) : list type :=
  List.map (subst_type_var_in_type sub) tl5.

Fixpoint  subst_nam_in_term (nm : nam) (a5 : atom) (t5 : term) {struct t5} : term :=
match t5 with
 | term_Nam nam5 ⇒ term_Nam (subst_nam_in_nam nm a5 nam5)
 | term_Var_b nat ⇒ term_Var_b nat
 | term_Var_f x ⇒ term_Var_f x
 | term_Ctor cfg n tl xl ⇒ term_Ctor cfg n tl (List.map (subst_nam_in_term nm a5) xl)
end.
Definition  subst_nam_in_term_list (nm : nam) (a : atom) (xl : list term) : list term :=
  List.map (subst_nam_in_term nm a) xl.

Fixpoint  subst_term_in_term (t5 : term) (x5 : atom) (t_6 : term) {struct t_6} : term :=
match t_6 with
 | term_Nam nam5 ⇒ term_Nam nam5
 | term_Var_b nat ⇒ term_Var_b nat
 | term_Var_f x ⇒ (if eq_var x x5 then t5 else (term_Var_f x))
 | term_Ctor cfg n tl xl ⇒ term_Ctor cfg n tl (List.map (subst_term_in_term t5 x5) xl)
end.
Definition  subst_term_in_term_list (t : term) (x : atom) (xl : list term) : list term :=
  List.map (subst_term_in_term t x) xl.

Fixpoint  subst_type_var_in_term (sub : list (string × type)) (t5 : term)
                                 {struct t5} : term :=
match t5 with
 | term_Nam nam5 ⇒ term_Nam nam5
 | term_Var_b nat ⇒ term_Var_b nat
 | term_Var_f x ⇒ term_Var_f x
 | term_Ctor cfg n tl xl ⇒ term_Ctor cfg n (subst_type_var_in_type_list sub tl)
                                    (List.map (subst_type_var_in_term sub ) xl)
end.
Definition  subst_type_var_in_term_list (sub : list (string × type))
                                    (xl5 : list term) : list term :=
  List.map (subst_type_var_in_term sub ) xl5.

Fixpoint  eq_var_in_nam (n m : nam) : Prop :=
match n, m with
 | Nam_f a, Nam_f b ⇒ if eq_var a b then True else False
 | _, _ ⇒ False
end.

Lemma  eq_var_in_nam_dec : ∀ x y, {eq_var_in_nam x y} + {¬ eq_var_in_nam x y}.

Fixpoint  subst_nam_with_term_in_term (t5 : term) (nm : nam) (t : term)
                                   {struct t} : term :=
match t with
 | term_Nam nam5 ⇒ (if eq_var_in_nam_dec nam5 nm then t5 else (term_Nam nam5))
 | term_Var_b nat ⇒ term_Var_b nat
 | term_Var_f x ⇒ term_Var_f x
 | term_Ctor cfg n tl xl ⇒ term_Ctor cfg n tl
                              (List.map (subst_nam_with_term_in_term t5 nm) xl)
end.
Definition  subst_nam_with_term_in_term_list (t5 : term) (nam5 : nam)
```

```
                                          (xl5 : list term) : list term :=
  List.map (subst_nam_with_term_in_term t5 nam5) xl5.

Definition  subst_nam_in_dtor (nam5 : nam) (a5 : atom) (g5 : dtor) : dtor :=
match g5 with
 | dtor_Dtor cfg n tl xl ⇒ dtor_Dtor cfg n tl (subst_nam_in_term_list nam5 a5 xl)
end.

Definition  subst_term_in_dtor (t5 : term) (x5 : atom) (g5 : dtor) : dtor :=
match g5 with
 | dtor_Dtor cfg n tl xl ⇒ dtor_Dtor cfg n tl (subst_term_in_term_list t5 x5 xl)
end.

Definition  subst_type_var_in_dtor (sub : list (string × type)) (g5 : dtor) : dtor :=
match g5 with
 | dtor_Dtor cfg n tl xl ⇒ dtor_Dtor cfg n (subst_type_var_in_type_list sub tl)
                                    (subst_type_var_in_term_list sub xl)
end.

Definition  subst_nam_with_term_in_dtor (t : term) (nm : nam) (g : dtor) : dtor :=
match g with
 | dtor_Dtor cf n tl xl ⇒ dtor_Dtor cf n tl (subst_nam_with_term_in_term_list t nm xl)
end.

Fixpoint  subst_nam_in_proc (nm : nam) (a : atom) (P5 : proc) {struct P5} : proc :=
match P5 with
 | proc_out t u P ⇒ proc_out (subst_nam_in_term nm a t) (subst_nam_in_term nm a u)
                             (subst_nam_in_proc nm a P)
 | proc_in t P ⇒ proc_in (subst_nam_in_term nm a t) (subst_nam_in_proc nm a P)
 | proc_bangin t P ⇒ proc_bangin (subst_nam_in_term nm a t) (subst_nam_in_proc nm a P)
 | proc_let g P Q ⇒ proc_let (subst_nam_in_dtor nm a g) (subst_nam_in_proc nm a P)
                             (subst_nam_in_proc nm a Q)
 | proc_new T P ⇒ proc_new T (subst_nam_in_proc nm a P)
 | proc_fork P Q ⇒ proc_fork (subst_nam_in_proc nm a P) (subst_nam_in_proc nm a Q)
 | proc_null ⇒ proc_null
end.

Fixpoint  subst_term_in_proc (t5 : term) (x5 : atom) (P5 : proc) {struct P5} : proc :=
match P5 with
 | proc_out t u P ⇒ proc_out (subst_term_in_term t5 x5 t) (subst_term_in_term t5 x5 u)
                             (subst_term_in_proc t5 x5 P)
 | proc_in t P ⇒ proc_in (subst_term_in_term t5 x5 t) (subst_term_in_proc t5 x5 P)
 | proc_bangin t P ⇒ proc_bangin (subst_term_in_term t5 x5 t)
                                 (subst_term_in_proc t5 x5 P)
 | proc_let g P Q ⇒ proc_let (subst_term_in_dtor t5 x5 g) (subst_term_in_proc t5 x5 P)
                             (subst_term_in_proc t5 x5 Q)
 | proc_new T P ⇒ proc_new T (subst_term_in_proc t5 x5 P)
 | proc_fork P Q ⇒ proc_fork (subst_term_in_proc t5 x5 P) (subst_term_in_proc t5 x5 Q)
 | proc_null ⇒ proc_null
end.

Definition  subst_type_var_in_fun_type (sub : list (string × type))
                                    (Tfun5 : fun_type) : fun_type :=
match Tfun5 with
```

```
 | ftype sl tl T ⇒ ftype sl (subst_type_var_in_type_list sub tl)
                            (subst_type_var_in_type sub T)
end.

Fixpoint  subst_type_var_in_proc (sb : list (string × type)) (P5 : proc)
                                 {struct P5} : proc :=
match P5 with
 | proc_out t u P ⇒ proc_out (subst_type_var_in_term sb t)
                            (subst_type_var_in_term sb u)
                            (subst_type_var_in_proc sb P)
 | proc_in t P ⇒ proc_in (subst_type_var_in_term sb t) (subst_type_var_in_proc sb P)
 | proc_bangin t P ⇒ proc_bangin (subst_type_var_in_term sb t)
                                 (subst_type_var_in_proc sb P)
 | proc_let g P Q ⇒ proc_let (subst_type_var_in_dtor sb g)
                            (subst_type_var_in_proc sb P)
                            (subst_type_var_in_proc sb Q)
 | proc_new T P ⇒ proc_new (subst_type_var_in_type sb T)
                           (subst_type_var_in_proc sb P)
 | proc_fork P Q ⇒ proc_fork (subst_type_var_in_proc sb P)
                            (subst_type_var_in_proc sb Q)
 | proc_null ⇒ proc_null
end.

Fixpoint  subst_nam_with_term_in_proc (t5 : term) (nam5 : nam) (P5 : proc)
                                      {struct P5} : proc :=
match P5 with
 | proc_out t u P ⇒ proc_out (subst_nam_with_term_in_term t5 nam5 t)
                            (subst_nam_with_term_in_term t5 nam5 u)
                            (subst_nam_with_term_in_proc t5 nam5 P)
 | proc_in t P ⇒ proc_in (subst_nam_with_term_in_term t5 nam5 t)
                         (subst_nam_with_term_in_proc t5 nam5 P)
 | proc_bangin t P ⇒ proc_bangin (subst_nam_with_term_in_term t5 nam5 t)
                                 (subst_nam_with_term_in_proc t5 nam5 P)
 | proc_let g P Q ⇒ proc_let (subst_nam_with_term_in_dtor t5 nam5 g)
                            (subst_nam_with_term_in_proc t5 nam5 P)
                            (subst_nam_with_term_in_proc t5 nam5 Q)
 | proc_new T P ⇒ proc_new T (subst_nam_with_term_in_proc t5 nam5 P)
 | proc_fork P Q ⇒ proc_fork (subst_nam_with_term_in_proc t5 nam5 P)
                            (subst_nam_with_term_in_proc t5 nam5 Q)
 | proc_null ⇒ proc_null
end.

End ParametrizedByName.
```

A.2. The Error Monad Library

```
Section ErrorMonad.

Definition bind {X Y : Type} (f : X → option Y) (ox : option X) :=
match ox with
 | Some x ⇒ f x
 | _ ⇒ None
end.

Lemma bind_bind : ∀ (X Y Z : Type) (f : X → option Y) (g : Y → option Z)
                    (x : option X),
  bind g (bind f x) = bind (fun u : X ⇒ bind g (f u)) x.

Lemma bind_some : ∀ (X Y : Type) (f : X → option Y) (x : X),
  bind f (Some x) = f x.

Lemma some_bind : ∀ (X : Type) (x : option X),
  bind (@Some X) x = x.

End ErrorMonad.


(* The Error Monad *)
Definition Maybe := Build_Monad option (@bind) Some bind_bind bind_some some_bind.

Definition munit {X : Type} := @Monad.unit Maybe X.
Definition mmap {X Y : Type} := @Monad.map Maybe X Y.
Definition mbind {X Y : Type} := @Monad.bind Maybe X Y.

Fixpoint unmaybe {X : Type} (xs : list (Maybe X)) : Maybe (list X) :=
match xs with
 | nil ⇒ Some nil
 | (Some x) :: ys ⇒ match unmaybe ys with
                     | Some zs ⇒ Some (x :: zs)
                     | _ ⇒ None
                    end
 | _ ⇒ None
end.
```

A.3. VPJ: Subtyping

```
(* Empty type environment *)
Definition  delta_empty : delta :=
  fun_empty _ _.
Definition  delta_add d X vT : delta :=
  fun_update StringDec.eq_dec d X (Some vT).
Definition  delta_one X vT : delta :=
  delta_add delta_empty X vT.
Definition  delta_del d X : delta :=
  fun_update StringDec.eq_dec d X None.
Definition  delta_append d1 d2 : delta :=
fun x ⇒ match (d1 x) with
          | Some vT ⇒ Some vT
          | None ⇒ d2 x
         end.
Definition  delta_init xvts : delta :=
  fun_init StringDec.eq_dec xvts.

Inductive  fresh_tvar : tvname → jtype → Prop :=
 |  fresh_tv : ∀ X Y,
              X ≠ Y →
              fresh_tvar X (TVar Y)
 |  fresh_tc : ∀ X cn ps,
              (∀ v T, In (v, T) ps → fresh_tvar X (RefType T)) →
              fresh_tvar X (RefType (TC (TClass cn ps))).

Inductive  opens_to : delta → jtype → delta → jtype → Prop :=
 |  open_refl : ∀ d T, opens_to d T delta_empty T
 |  open_class : ∀ d d' cn ps ps',
               opens_to_list d ps d' ps' →
               opens_to d (RefType (TC (TClass cn ps)))
                        d' (RefType (TC (TClass cn ps')))
with  opens_to_list : delta → list (jvariance × jparam_type) → delta →
                      list (jvariance × jparam_type) → Prop :=
 |  open_to_nil : ∀ d d', opens_to_list d nil d' nil
 |  open_to_inv : ∀ d d' T vts vus,
                opens_to_list d vts d' vus →
                opens_to_list d ((JVInv, T) :: vts) d' ((JVInv, T) :: vus)
 |  open_to_list : ∀ d d' v X T vts vus,
                 v ≠ JVInv →
                 d X = None →
                 (∀ Y v U, d Y = Some (v, U) → fresh_tvar X (RefType U)) →
                 d' X = None →
                 (∀ Y v U, d' Y = Some (v, U) → fresh_tvar X (RefType U)) →
                 fresh_tvar X (RefType T) →
                 (∀ v U, In (v, U) vts → fresh_tvar X (RefType U)) →
                 (∀ v U, In (v, U) vus → fresh_tvar X (RefType U)) →
                 opens_to_list d vts d' vus →
                 opens_to_list d ((v, T) :: vts)
                               (delta_add d' X (v, T)) ((JVInv, TV X) :: vus).
```

```
Inductive  closes_to : jtype → delta → jtype → Prop :=
 |  close_void : ∀ d, closes_to Void d Void
 |  close_bool : ∀ d, closes_to Boolean d Boolean
 |  close_int : ∀ d, closes_to Integer d Integer
 |  close_str : ∀ d, closes_to String d String
 |  close_null : ∀ d, closes_to NullType d NullType
 |  close_prom : ∀ X d T,
                 d X = Some (JVCov, T) →
                 closes_to (TVar X) d (RefType T)
 |  close_tvar : ∀ X d,
                 d X = None →
                 closes_to (TVar X) d (TVar X)
 |  close_class_refl : ∀ cn d vTs wTs,
                       closes_to_list vTs d wTs →
                       closes_to (Class cn vTs) d (Class cn wTs)
with  closes_to_list : list (jvariance × jparam_type) → delta →
                       list (jvariance × jparam_type) → Prop :=
 |  close_nil : ∀ d, closes_to_list nil d nil
 |  close_list_refl : ∀ v T d vTs wTs,
                      closes_to (RefType T) d (RefType T) →
                      closes_to_list vTs d wTs →
                      closes_to_list ((v, T) :: vTs) d ((v, T) :: wTs)
 |  close_list_type : ∀ v T U d vTs wTs,
                      closes_to (RefType T) d (RefType U) →
                      T ≠ U →
                      closes_to_list vTs d wTs →
                      closes_to_list ((v, T) :: vTs) d ((varlub v JVCov, U) :: wTs)
 |  close_list_tvar : ∀ v w X U d vTs wTs,
                      d X = Some (w, U) →
                      closes_to_list vTs d wTs →
                      closes_to_list ((v, TV X) :: vTs) d ((varlub v w, U) :: wTs).

Definition  subtypes {E : Type} (P : @prog E) (d : delta) (ts : list jtype)
                                (us : list jtype) : Prop :=
  fold_right (fun x p ⇒ subtype P d (fst x) (snd x) ∧ p) True (combine ts us).
```

A.4. VPJ: Expression Typing

```
Section  Java.

(* Instantiated with expression on use (needed to break circular dependency) *)
Variable  E : Type.

Inductive  WT_binop : @prog E → jtype → bop → jtype → jtype → Prop :=
 |  WT_binop_Eq : ∀ P d T1 T2,
                  subtype P d T1 T2 ∨ subtype P d T2 T1 →
                  WT_binop P T1 Eq T2 Boolean
 |  WT_binop_NotEq : ∀ P d T1 T2,
                     subtype P d T1 T2 ∨ subtype P d T2 T1 →
                     WT_binop P T1 NotEq T2 Boolean
 |  WT_binop_LessThan : ∀ P, WT_binop P Integer LessThan Integer Boolean
 |  WT_binop_LessOrEqual : ∀ P, WT_binop P Integer LessOrEqual Integer Boolean
 |  WT_binop_GreaterThan : ∀ P, WT_binop P Integer GreaterThan Integer Boolean
 |  WT_binop_GreaterOrEqual : ∀ P,
                     WT_binop P Integer GreaterOrEqual Integer Boolean
 |  WT_binop_Add : ∀ P, WT_binop P Integer Add Integer Integer
 |  WT_binop_Subtract : ∀ P, WT_binop P Integer Subtract Integer Integer
 |  WT_binop_Mult : ∀ P, WT_binop P Integer Mult Integer Integer
 |  WT_binop_BinAnd : ∀ P, WT_binop P Boolean BoolAnd Boolean Boolean
 |  WT_binop_BinOr : ∀ P, WT_binop P Boolean BoolOr Boolean Boolean
 |  WT_binop_BinXor : ∀ P, WT_binop P Boolean BoolXor Boolean Boolean.

End Java.

Section Java.

(* Instantiated with expression on use (needed to break circular dependency) *)
Variable E : Type.

Inductive  external_WT : @prog E → jtype → mname → list jtype →
                         jtype → Prop :=
 |  WTNewThread : ∀ P C,
                  subclass P C Thread →
                  external_WT P (RefType (TC (TClass C nil))) start nil Void
 |  WTWait : ∀ P T, external_WT P (RefType T) wait nil Void
 |  WTNotify : ∀ P T, external_WT P (RefType T) notify nil Void
 |  WTNotifyAll : ∀ P T, external_WT P (RefType T) notifyAll nil Void
 |  WTJoin : ∀ P C,
             subclass P C Thread →
             external_WT P (RefType (TC (TClass C nil))) join nil Void.

End Java.

Section  WellFoundedSuperclass.

(* Well-formedness assumption *)
Hypothesis  subclass1_wf : well_founded (superclass1).

Inductive  is_mtype : jclass_type →
                      mname →
```

```
                    list (tvname × jparam_type) →
                    list jtype →
                    jtype →
                    Prop :=
 | mt_class : ∀ C m cn ps Ts T fns (fd : fdecl) (md : mdecl) tvts mtvnts vnts e
                ctvnts D tret,
              P cn = Some (ctvnts, D, (fns, fd), md) →
              md m = Some (mtvnts, vnts, tret, e) →
              C = InvTClass cn ps →
              tvts = (map (fun x ⇒ (fst x, subst_jtvars_in_jparam
                            (map fst ctvnts) ps (snd x))) mtvnts) →
              Ts = (map (subst_jtvars_in_jtype (map fst ctvnts) ps)
                        (map snd vnts)) →
              T = subst_jtvars_in_jtype (map fst ctvnts) ps tret →
              is_mtype C m tvts Ts T
 | mt_super : ∀ C m cn ps tvts Ts T ctvnts D fns (fd : fdecl) (md : mdecl),
              P cn = Some (ctvnts, D, (fns, fd), md) →
              md m = None →
              C = InvTClass cn ps →
              is_mtype (subst_jtvars_in_jclass (map fst ctvnts) ps D)
                       m tvts Ts T →
              is_mtype C m tvts Ts T.

Function fields_of (cn : cname) {wf (superclass1) cn} : list (vname × cname) :=
match (P cn) with
 | Some (_, TClass dn ps, (fns, _), _)
     ⇒ if StringDec.eq_dec cn Object then
            nil
        else
            (fields_of dn) ++ (map (fun v ⇒ (v, cn)) fns)
 | None ⇒ nil
end.

Inductive  is_ftype : jclass_type →
                      vname →
                      jtype →
                      Prop :=
 | is_ftype_object : ∀ C fn U ctvnts D fns fd md,
                     P Object = Some (ctvnts, D, (fns, fd), md) →
                     fd fn = Some U →
                     C = InvTClass Object nil →
                     is_ftype C fn U
 | is_ftype_class : ∀ C cn ps fn U U' ctvnts D fns fd md,
                    P cn = Some (ctvnts, D, (fns, fd), md) →
                    fd fn = Some U →
                    C = InvTClass cn ps →
                    U' = (subst_jtvars_in_jtype (map fst ctvnts) ps U) →
                    is_ftype C fn U'
 | is_ftype_super : ∀ C cn ps fn U ctvnts D fns fd md,
                    P cn = Some (ctvnts, D, (fns, fd), md) →
                    fd fn = None →
                    C = InvTClass cn ps →
```

```
                  is_ftype (subst_jtvars_in_jclass (map fst ctvnts) ps D) fn U →
                  is_ftype C fn U.

End WellFoundedSuperclass.

Definition  gamma_empty : gamma :=
  fun_empty _ _.
Definition  gamma_add g x T : gamma :=
  fun_update StringDec.eq_dec g x (Some T).
Definition  gamma_update g x oT : gamma :=
  fun_update StringDec.eq_dec g x oT.
Definition  gamma_init xts : gamma :=
  fun_init StringDec.eq_dec xts.

Inductive  is_bound_of : delta → jtype → jtype → Prop :=
 |  bound_void : ∀ d, is_bound_of d Void Void
 |  bound_bool : ∀ d, is_bound_of d Boolean Boolean
 |  bound_int : ∀ d, is_bound_of d Integer Integer
 |  bound_str : ∀ d, is_bound_of d String String
 |  bound_null : ∀ d, is_bound_of d NullType NullType
 |  bound_tvar : ∀ d X T U,
                  d X = Some (JVCov, T) →
                  is_bound_of d (RefType T) U →
                  is_bound_of d (RefType (TV X)) U
 |  bound_class : ∀ d C,
                   is_bound_of d (RefType (TC C)) (RefType (TC C)).

Inductive is_wt_expr : jprog → delta → gamma → expr → jtype → Prop :=
 |  wte_new : ∀ P d g cn ps eargs Us fncs
               (Hwf : well_founded (superclass1 P)),
               fncs = fields_of P Hwf cn →
               is_wf_type P d (RefType (TC (TClass cn ps))) →
               length fncs = length eargs →
               (∀ fn cn' U Tf, In ((fn, cn'), U) (combine fncs Us) →
                  is_ftype P (TClass cn ps) fn Tf →
                  subtype P delta_empty U Tf) →
               are_wt_exprs P d g eargs Us →
               is_wt_expr P d g (New (TClass cn ps) eargs)
                                (RefType (TC (TClass cn ps)))
 |  wte_cast : ∀ P d g T U boundT boundU e,
                is_wt_expr P d g e T →
                is_wf_type P d U →
                is_bound_of d T boundT →
                is_bound_of d U boundU →
                (subtype P d boundT boundU ∨ subtype P d boundU boundT) →
                is_wt_expr P d g (Cast U e) U
 |  wte_scast : ∀ P d g T U boundT boundU e,
                 is_wt_expr P d g e T →
                 is_wf_type P d U →
                 is_bound_of d T boundT →
                 is_bound_of d U boundU →
                 ¬ (subtype P d boundT boundU) →
                 ¬ (subtype P d boundU boundT) →
```

```
                is_wt_expr P d g (Cast U e) U
| wte_val : ∀ P d g v T,
              typeof v = Some T →
              is_wt_expr P d g (Val v) T
| wte_binop : ∀ P d g op e1 e2 T1 T2 U,
                is_wt_expr P d g e1 T1 →
                is_wt_expr P d g e2 T2 →
                WT_binop P T1 op T2 U →
                is_wt_expr P d g (BinOp e1 op e2) U
| wte_var : ∀ P d g x T,
              wf_gamma P d g →
              g x = Some T →
              is_wt_expr P d g (Var x) T
| wte_lass : ∀ P d g x e (T : jtype) boundT U boundU,
               x ≠ this →
               g x = Some U →
               U ≠ Void →
               is_bound_of d U boundU →
               is_wt_expr P d g e T →
               is_bound_of d T boundT →
               subtype P d boundT boundU →
               is_wt_expr P d g (LAss x e) Void
| wte_facc : ∀ P d d' g e fn cn ps T boundT (Tf : jtype) U,
               is_wt_expr P d g e T →
               is_bound_of d T boundT →
               opens_to d boundT d' (Class cn ps) →
               Tf ≠ Void →
               Tf ≠ NullType →
               is_ftype P (TClass cn ps) fn Tf →
               closes_to Tf d' U →
               is_wt_expr P d g (FAcc e fn cn) U
| wte_fass : ∀ P d d' g e1 e2 fn cn ps (Tf : jtype) T boundT U boundU V boundV,
               is_wt_expr P d g e1 T →
               is_bound_of d T boundT →
               opens_to d boundT d' (Class cn ps) →
               Tf ≠ Void →
               Tf ≠ NullType →
               is_ftype P (TClass cn ps) fn Tf →
               closes_to Tf d' U →
               is_bound_of d U boundU →
               is_wt_expr P d g e2 V →
               is_bound_of d V boundV →
               subtype P d boundV boundU →
               is_wt_expr P d g (FAss e1 fn cn e2) Void
| wte_call : ∀ P d d' g e tvns tvts mn eargs (Ts Us : list jtype) cn ps T
               boundT (Tm Tret : jtype) tvtargs,
               is_wt_expr P d g e T →
               ¬ is_external_call P T mn →
               is_bound_of d T boundT →
               opens_to d boundT d' (Class cn ps) →
               is_mtype P (TClass cn ps) mn (combine tvns tvts) Ts Tret →
               (∀ X, In X tvns → d' X = None) →
```

```
              (∀ T, In T tvtargs → is_wf_type P d (RefType T)) →
              subtypes P (delta_append d d') (map RefType tvtargs)
                       (map (fun x ⇒ RefType (subst_jtvars_in_jparam
                                                tvns tvtargs x)) tvts) →
              are_wt_exprs P d g eargs Us →
              subtypes P (delta_append d d') Us
                         (map (subst_jtvars_in_jtype tvns tvtargs) Ts) →
              closes_to (subst_jtvars_in_jtype tvns tvtargs Tret) d' Tm →
              is_wt_expr P d g (Call e tvtargs mn eargs) Tm
| wte_external : ∀ P d g e cn mn eargs (Ts : list jtype) (T : jtype),
                  is_wt_expr P d g e (Class cn nil) →
                  is_external_call P (Class cn nil) mn →
                  external_WT P (Class cn nil) mn Ts T →
                  are_wt_exprs P d g eargs Ts →
                  is_wt_expr P d g (Call e nil mn eargs) T
| wte_block : ∀ P d g x T ov v e U,
               is_wf_type P d T →
               (ov = None ∨ (ov = Some v
                             ∧ ∃ T', typeof v = Some T'
                             ∧ subtype P d T' T)) →
               is_wt_expr P d (gamma_add g x T) e U →
               is_wt_expr P d g (Block x T ov e) U
| wte_sync : ∀ P d g eex e T U,
              is_wt_expr P d g eex (RefType T) →
              is_wt_expr P d g e U →
              is_wt_expr P d g (jsync eex e) U
| wte_seq : ∀ P d g e1 e2 T1 T2,
             is_wt_expr P d g e1 T1 →
             is_wt_expr P d g e2 T2 →
             is_wt_expr P d g (Seq e1 e2) T2
| wte_cond : ∀ P d g c e1 e2 T1 T2 boundT1 boundT2 T,
              is_wt_expr P d g c Boolean →
              is_wt_expr P d g e1 T1 →
              is_wt_expr P d g e2 T2 →
              is_bound_of d T1 boundT1 →
              is_bound_of d T2 boundT2 →
              (subtype P d boundT1 boundT2 ∨ subtype P d boundT2 boundT1) →
              (subtype P d boundT1 boundT2 → T = T2) →
              (subtype P d boundT2 boundT1 → T = T1) →
              is_wt_expr P d g (Cond c e1 e2) T
| wte_while : ∀ P d g c e T,
               is_wt_expr P d g c Boolean →
               is_wt_expr P d g e T →
               is_wt_expr P d g (while (c) e) Void
| wte_throw : ∀ P d g ex Tex,
               is_wt_expr P d g ex Tex →
               is_instance_type Tex →
               subtype P d Tex TThrowable →
               is_wt_expr P d g (throw ex) Void
| wte_trycatch : ∀ P d g e1 e2 Tex x T,
                  is_wt_expr P d g e1 T →
                  subtype P d (RefType (TC Tex)) TThrowable →
```

```
                  is_wt_expr P d (gamma_add g x (RefType (TC Tex))) e2 T →
                  is_wt_expr P d g (TryCatch (e1) Tex x (e2)) T
with  are_wt_exprs : jprog → delta → gamma → list expr → list jtype → Prop :=
 |  wte_nil : ∀ P d g, are_wt_exprs P d g nil nil
 |  wte_cons : ∀ P d g e es T Ts,
               is_wt_expr P d g e T →
               are_wt_exprs P d g es Ts →
               are_wt_exprs P d g (e :: es) (T :: Ts).

Definition  overrides (J : jprog)
                      (C : jclass_type)
                      (mn : mname)
                      (Ys : list tvname)
                      (Ps : list jparam_type)
                      (Ts : list jtype)
                      (T : jtype) : Prop :=
  ∀ Zs Qs Us U,
    is_mtype J C mn (combine Zs Qs) Us U →
    length Zs = length Qs →
    length Ys = length Ps →
      (Ps = (map (subst_jtvars_in_jparam Zs (map TV Ys)) Qs)
       ∧ Ts = (map (subst_jtvars_in_jtype Zs (map TV Ys)) Us)
       ∧ T = (subst_jtvars_in_jtype Zs (map TV Ys) U)).

Inductive  has_standard_classes : jprog → Prop :=
 |  hsc_rule : ∀ P (objfd thrfd trwfd npfd ccfd oomfd imsfd itsfd : fdecl)
                   (objmd thrmd trwmd npmd ccmd oommd imsmd itsmd : mdecl),
               (* Object is not parametrized, is a subclass of itself and has
                  neither fields nor methods *)
               P Object = Some (nil, CObject, (nil, objfd), objmd) →
               (∀ fn, objfd fn = None) →
               (∀ mn, objmd mn = None) →
               (* Thread is not parametrized, is a subclass of Object, has no fields
                  and one method "run" *)
               P Thread = Some (nil, CObject, (nil, thrfd), thrmd) →
               (∀ fn, thrfd fn = None) →
               (thrmd run = Some (nil, nil, Void, junit)
                 ∧ (∀ mn, mn ≠ run → thrmd mn = None)) →
               P Throwable = Some (nil, CObject, (nil, trwfd), trwmd) →
               (∀ fn, trwfd fn = None) →
               (∀ mn, trwmd mn = None) →
               (* system exceptions are subclasses of Throwable *)
               P "NullPointer" = Some (nil, CThrowable, (nil, npfd), npmd) →
               (∀ fn, npfd fn = None) →
               (∀ mn, npmd mn = None) →
               P "ClassCast" = Some (nil, CThrowable, (nil, ccfd), ccmd) →
               (∀ fn, ccfd fn = None) →
               (∀ mn, ccmd mn = None) →
               P "OutOfMemory" = Some (nil, CThrowable, (nil, oomfd), oommd) →
               (∀ fn, oomfd fn = None) →
               (∀ mn, oommd mn = None) →
               P "IllegalMonitorState" = Some (nil, CThrowable, (nil, imsfd), imsmd) →
```

```
(∀ fn, imsfd fn = None) →
(∀ mn, imsmd mn = None) →
P "IllegalThreadState" = Some (nil, CThrowable, (nil, itsfd), itsmd) →
(∀ fn, itsfd fn = None) →
(∀ mn, itsmd mn = None) →
has_standard_classes P.
```

A.5. VPJ: Small-Step Semantics

```
Fixpoint  typeof_h (h : heap) (v : val) : Maybe jtype :=
match v with
 | Unit ⇒ Some Void
 | Null ⇒ Some NullType
 | Bool _ ⇒ Some Boolean
 | Intg _ ⇒ Some Integer
 | Str _ ⇒ Some String
 | Addr a ⇒ match h a with
              | Some (Obj C fs) ⇒ Some (RefType (TC C))
              | None ⇒ None
            end
end.

Fixpoint  binop (op : bop) (a b : val) : Maybe val :=
match op, a, b with
 | Eq, v1, v2 ⇒ Some (Bool (bool_of (ValType.eq_dec v1 v2)))
 | NotEq, v1, v2 ⇒ Some (Bool (negb (bool_of (ValType.eq_dec v1 v2))))
 | LessThan, (Intg i1), (Intg i2) ⇒ Some (Bool (Zlt i1 i2))
 | LessOrEqual, (Intg i1), (Intg i2) ⇒ Some (Bool (Zle i1 i2))
 | GreaterThan, (Intg i1), (Intg i2) ⇒ Some (Bool (Zgt i1 i2))
 | GreaterOrEqual, (Intg i1), (Intg i2) ⇒ Some (Bool (Zge i1 i2))
 | Add, (Intg i1), (Intg i2) ⇒ Some (Intg (i1 + i2))
 | Subtract, (Intg i1), (Intg i2) ⇒ Some (Intg (i1 - i2))
 | Mult, (Intg i1), (Intg i2) ⇒ Some (Intg (i1 × i2))
 | BoolAnd, (Bool v1), (Bool v2) ⇒ Some (Bool (andb v1 v2))
 | BoolOr, (Bool v1), (Bool v2) ⇒ Some (Bool (orb v1 v2))
 | BoolXor, (Bool v1), (Bool v2) ⇒ Some (Bool (xorb v1 v2))
 | _, _, _ ⇒ None
end.

Section  MethodLookup.


(* The VPJ program, all definitions below are parameterized by it *)
Variable  P : jprog.
(* Well-formedness assumption *)
Hypothesis  subclass_wf : well_founded (superclass1 P).

Definition  subst_and_convert_args (xs : list tvname)
                                   (ts : list jparam_type)
                                   (vnts : list (vname × jtype))
                                   : list jtype :=
  map (subst_jtvars_in_jtype xs ts) (map snd vnts).

Definition  subst_and_convert_mbody (xs : list tvname)
                                    (ts : list jparam_type)
                                    (D_md : Maybe (jclass_type × jmdef))
                                    : Maybe (jclass_type × list vname × list jtype
                                            × jtype × expr) :=
match D_md with
 | Some (D, (tvnts, vnts, T, e)) ⇒ Some (subst_jtvars_in_jclass xs ts D,
```

```
                                        map fst vnts,
                                        subst_and_convert_args xs ts vnts,
                                        subst_jtvars_in_jtype xs ts T,
                                        subst_jtvars_in_expr xs ts e)
 | None ⇒ None
end.

Definition  subst_mbody (xs : list tvname)
                        (ts : list jparam_type)
                        (mb : Maybe (jclass_type × list vname × list jtype
                                    × jtype × expr))
                        : Maybe (jclass_type × list vname × list jtype
                                × jtype × expr) :=
match mb with
 | Some (D, vns, Ts, T, e) ⇒ Some (subst_jtvars_in_jclass xs ts D,
                                   vns,
                                   map (subst_jtvars_in_jtype xs ts) Ts,
                                   subst_jtvars_in_jtype xs ts T,
                                   subst_jtvars_in_expr xs ts e)
 | None ⇒ None
end.

Function mbody' (cn : cname)
                (ctvts : list jparam_type)
                (m : mname)
                (mtvts : list jparam_type)
                {wf (superclass1 P) cn} : Maybe (jclass_type
                                                 × (list vname)
                                                 × (list jtype)
                                                 × jtype
                                                 × expr) :=
match (P cn) with
 | Some (ctvnts, TClass dn ps, _, mds) ⇒
     match (mds m) with
      | Some (mtvnts, vnts, T, e)
              ⇒ subst_mbody (map fst ctvnts) ctvts (subst_and_convert_mbody
                            (map fst mtvnts) mtvts (Some (TClass dn ps,
                                                          (mtvnts, vnts, T, e))))
      | None ⇒ if StringDec.eq_dec cn Object then
                  None
                else
                  mbody' dn (map (subst_jtvars_in_jparam (map fst ctvnts) ctvts)
                                 (map snd ps)) m mtvts
     end
 | None ⇒ None
end.

Definition  mbody (C : jclass_type)
                  (m : mname)
                  (mtvts : list jparam_type)
                  : Maybe (jclass_type × (list vname)
                          × (list jtype) × jtype × expr) :=
match C with
```

```
 | TClass cn cps ⇒ mbody' cn (map snd cps) m mtvts
end.

End MethodLookup.

Section Java.

(* The VPJ program, all definitions below are parameterized by it *)
Variable P : jprog.
(* Well-formedness assumption *)
Hypothesis subclass_wf : well_founded (superclass1 P).


Fixpoint  blocks (la : list vname) (lt : list jtype) (lv : list val)
                 (e : expr) : expr :=
match (la, lt, lv) with
 | (V :: Vs, T :: Ts, v :: vs) ⇒ Block V T (Some v) (blocks Vs Ts vs e)
 | (nil, nil, nil) ⇒ e
 | (_, _, _) ⇒ e
end.


Definition  J_thread_action := thread_action (expr × locals) heap (Maybe obs_event).

Definition  extNTA2J : (cname × mname × addr) → (expr × locals) :=
  fun x ⇒
  match x with (C, m, a) ⇒
    match (mbody' P subclass_wf C nil m nil) with
     | Some (D, vns, Ts, T, e) ⇒ (Block this (RefType (TC D))
                                             (Some (Addr a)) e, stack_empty)
     | _ ⇒ (junit, stack_empty)
    end
  end.

Definition  extTA2J : external_thread_action → J_thread_action :=
  convert_extTA (extNTA2J).


Fixpoint  extRet2J (x : val + addr) : expr :=
match x with
 | inl v ⇒ Val v
 | inr a ⇒ jThrow a
end.

Inductive red : (external_thread_action → J_thread_action) →
                (expr × (heap × locals)) →
                J_thread_action →
                (expr × (heap × locals)) → Prop :=
 |  RedNew : ∀ extTA h a cn tvnts D fns fds mds h' l (args : list val) fncs,
             fncs = fields_of P subclass_wf cn →
             length fncs = length args →
             new_Addr h = Some a →
             P cn = Some (tvnts, D, (fns, fds), mds) →
             h' = heap_update h a (Some (Obj (TClass cn nil)
                    (fun_init StringPairDec.eq_dec (combine fncs args)))) →
             red extTA ((New (InvTClass cn (map TC (map snd tvnts))) (map Val args)),
```

```
                    (h, l)) empty_ta ((jaddr a), (h', l))
| RedNewFail : ∀ extTA s C args,
              new_Addr (hp s) = None →
              red extTA ((New C args), s) empty_ta ((jTHROW eOutOfMemory), s)
| NewArgs : ∀ extTA C es s ta es' s',
           reds extTA es s ta es' s' →
           red extTA ((New C (es)), s) ta ((New C (es')), s')
| CastRed : ∀ extTA e s ta e' s' C,
           red extTA (e, s) ta (e', s') →
           red extTA ((Cast C e), s) ta ((Cast C e'), s')
| RedCast : ∀ extTA s v T U,
           typeof_h (hp s) v = Some U →
           subtype P delta_empty U T →
           red extTA ((Cast T (Val v)), s) empty_ta ((Val v), s)
| RedCastFail : ∀ extTA d s v T U,
               typeof_h (hp s) v = Some U →
               ¬ subtype P d U T →
               red extTA ((Cast T (Val v)), s) empty_ta ((jTHROW eClassCast), s)
| BinOpRed1 : ∀ extTA e s ta e' s' op e2,
             red extTA (e, s) ta (e', s') →
             red extTA ((BinOp e op e2), s) ta ((BinOp e' op e2), s')
| BinOpRed2 : ∀ extTA e s ta e' s' v op,
             red extTA (e, s) ta (e', s') →
             red extTA ((BinOp (Val v) op e), s) ta ((BinOp (Val v) op e'), s')
| RedBinOp : ∀ extTA op v1 v2 v s,
            binop op v1 v2 = Some v →
            red extTA ((BinOp (Val v1) op (Val v2)), s) empty_ta ((Val v), s)
| RedVar : ∀ extTA s v V,
          lcl s V = Some v →
          red extTA ((Var V), s) empty_ta ((Val v), s)
| LAssRed : ∀ extTA e s ta e' s' V,
           red extTA (e, s) ta (e', s') →
           red extTA ((LAss V e), s) ta ((LAss V e'), s')
| RedLAss : ∀ extTA V v h l,
           red extTA ((LAss V (Val v)), (h, l)) empty_ta
               ((junit), (h, (stack_update l V (Some v))))
| FAccRed : ∀ extTA e s ta e' s' F D,
           red extTA (e, s) ta (e', s') →
           red extTA ((FAcc e F D), s) ta ((FAcc e' F D), s')
| RedFAcc : ∀ extTA s a C (fs : fields) F D v,
           hp s a = Some (Obj C fs) →
           fs (F, D) = Some v →
           red extTA ((FAcc (jaddr a) F D), s) empty_ta ((Val v), s)
| RedFAccNull : ∀ extTA F D s,
               red extTA ((FAcc jnull F D), s) empty_ta
                   ((jTHROW eNullPointer), s)
| FAssRed1 : ∀ extTA e s ta e' s' F D e2,
            red extTA (e, s) ta (e', s') →
            red extTA ((FAss e F D e2), s) ta ((FAss e' F D e2), s')
| FAssRed2 : ∀ extTA e s ta e' s' v F D,
            red extTA (e, s) ta (e', s') →
            red extTA ((FAss (Val v) F D e), s) ta ((FAss (Val v) F D e'), s')
```

```
| RedFAss : ∀ extTA a C fs F D v h l,
           h a = Some(Obj C fs) →
           red extTA ((FAss (jaddr a) F D (Val v)), (h, l)) empty_ta (junit,
               (heap_update h a (Some (Obj C (fields_update fs (F, D)
                                (Some v))))), l))
| RedFAssNull : ∀ extTA F D v s,
               red extTA ((FAss jnull F D (Val v)), s) empty_ta
                   ((jTHROW eNullPointer), s)
| CallObj : ∀ extTA e s ta e' s' tvts M es,
           red extTA (e, s) ta (e', s') →
           red extTA ((Call e tvts M (es)), s) ta
               ((Call e' tvts M (es)), s')
| CallArgs : ∀ extTA es s ta es' s' v M tvts,
            reds extTA es s ta es' s' →
            red extTA ((Call (Val v) tvts M (es)), s) ta
                ((Call (Val v) tvts M (es')), s')
| RedCall : ∀ extTA s a C fds M Ts T D args vns e tvts,
           hp s a = Some (Obj C fds) →
           ¬ is_external_call P (RefType (TC C)) M →
           mbody P subclass_wf C M tvts = Some (D, vns, Ts, T, e) →
           length args = length vns →
           length Ts = length vns →
           red extTA ((Call (jaddr a) tvts M (map Val args)), s) empty_ta
               ((blocks (this :: vns) ((RefType (TC D)) :: Ts)
                        ((Addr a) :: args) e), s)
| RedCallExternal : ∀ extTA s a T M vs va h' ta' ta e',
                   typeof_h (hp s) (Addr a) = Some T →
                   is_external_call P T M →
                   red_external P (hp s) a M vs ta va h' →
                   ta' = extTA ta →
                   e' = extRet2J va →
                   red extTA ((Call (jaddr a) nil M (map Val vs)), s) ta'
                       (e', (h', lcl s))
| RedCallNull : ∀ extTA M vs s tvts,
               red extTA ((Call jnull tvts M (map Val vs)), s) empty_ta
                   ((jTHROW eNullPointer), s)
| BlockRed : ∀ extTA e h l x vo ta e' h' l' T,
            red extTA (e, (h, stack_update l x vo)) ta (e', (h', l')) →
            red extTA ((Block x T vo e), (h, l)) ta
                ((Block x T (l' x) e'), (h', stack_update l' x (l x)))
| RedBlock : ∀ extTA V T vo u s,
            red extTA ((Block V T vo (Val u)), s) empty_ta ((Val u), s)
| SynchronizedRed1 : ∀ extTA o' s ta o'' s' e,
                    red extTA (o', s) ta (o'', s') →
                    red extTA ((jsync (o') e), s) ta ((jsync (o'') e), s')
| SynchronizedNull : ∀ extTA e s,
                    red extTA ((jsync (jnull) e), s) empty_ta
                        ((jTHROW eNullPointer), s)
| LockSynchronized : ∀ extTA s a arrobj e,
                    hp s a = Some arrobj →
                    red extTA ((jsync (jaddr a) e), s)
                        (ta_update_obs (ta_update_locks empty_ta Lock a)
```

```
                                        (Synchronization a)) ((jinsync (a) e), s)
| SynchronizedRed2 : ∀ extTA e s ta e' s' a,
                     red extTA (e, s) ta (e', s') →
                     red extTA ((jinsync (a) e), s) ta ((jinsync (a) e'), s')
| UnlockSynchronized : ∀ extTA a v s,
                       red extTA ((jinsync (a) (Val v)), s)
                           (ta_update_obs (ta_update_locks empty_ta Unlock a)
                                          (Synchronization a)) ((Val v), s)
| SeqRed : ∀ extTA e s ta e' s' e2,
           red extTA (e, s) ta (e', s') →
           red extTA ((Seq e e2), s) ta ((Seq e' e2), s')
| RedSeq : ∀ extTA v e s,
           red extTA ((Seq (Val v) e), s) empty_ta (e, s)
| CondRed : ∀ extTA b s ta b' s' e1 e2,
            red extTA (b, s) ta (b', s') →
            red extTA ((Cond (b) e1 e2), s) ta ((Cond (b') e1 e2), s')
| RedCondT : ∀ extTA e1 e2 s,
             red extTA ((Cond (jtrue) e1 e2), s) empty_ta (e1, s)
| RedCondF : ∀ extTA e1 e2 s,
             red extTA ((Cond (jfalse) e1 e2), s) empty_ta (e2, s)
| RedWhile : ∀ extTA b c s,
             red extTA ((while (b) c), s) empty_ta
                 ((Cond (b) (Seq c (while (b) c)) junit), s)
| ThrowRed : ∀ extTA e s ta e' s',
             red extTA (e, s) ta (e', s') →
             red extTA ((throw e), s) ta ((throw e'), s')
| RedThrowNull : ∀ extTA s,
                 red extTA ((throw jnull), s) empty_ta ((jTHROW eNullPointer), s)
| TryRed : ∀ extTA e s ta e' s' e2 C V,
           red extTA (e, s) ta (e', s') →
           red extTA ((TryCatch e C V e2), s) ta ((TryCatch e' C V e2), s')
| RedTry : ∀ extTA v C V e2 s,
           red extTA ((TryCatch (Val v) C V e2), s) empty_ta ((Val v), s)
| RedTryCatch : ∀ extTA d s a D fs C V e2,
                hp s a = Some(Obj D fs) →
                subtype P d (RefType (TC D)) (RefType (TC C)) →
                red extTA ((TryCatch (jThrow a) C V e2), s) empty_ta
                    ((Block V (RefType (TC C)) (Some (Addr a)) e2), s)
| RedTryFail : ∀ extTA d s a D fs C V e2,
               hp s a = Some(Obj D fs) →
               ¬ subtype P d (RefType (TC D)) (RefType (TC C)) →
               red extTA ((TryCatch (jThrow a) C V e2), s) empty_ta
                   ((jThrow a), s)
| CastThrow : ∀ extTA C a s,
              red extTA ((Cast C (jThrow a)), s) empty_ta ((jThrow a), s)
| BinOpThrow1 : ∀ extTA a e2 s op,
                red extTA ((BinOp (jThrow a) op e2), s) empty_ta
                    ((jThrow a), s)
| BinOpThrow2 : ∀ extTA v1 a s op,
                red extTA ((BinOp (Val v1) op (jThrow a)), s) empty_ta
                    ((jThrow a), s)
| LAssThrow : ∀ extTA V a s,
```

```
              red extTA ((LAss V (jThrow a)), s) empty_ta ((jThrow a), s)
 |  FAccThrow : ∀ extTA a F D s,
              red extTA ((FAcc (jThrow a) F D), s) empty_ta ((jThrow a), s)
 |  FAssThrow1 : ∀ extTA a F D e2 s,
               red extTA ((FAss (jThrow a) F D e2), s) empty_ta ((jThrow a), s)
 |  FAssThrow2 : ∀ extTA v F D a s,
               red extTA ((FAss (Val v) F D (jThrow a)), s) empty_ta
                   ((jThrow a), s)
 | CallThrowObj : ∀ extTA a M es s tvts,
                 red extTA ((Call (jThrow a) tvts M (es)), s) empty_ta
                     ((jThrow a), s)
 |  CallThrowParams : ∀ extTA es vs a es' v M s tvts,
                   es = map Val vs ++ ((jThrow a) :: es') →
                   red extTA ((Call (Val v) tvts M (es)), s) empty_ta
                       ((jThrow a), s)
 |  BlockThrow : ∀ extTA V T vo a s,
               red extTA ((Block V T vo (jThrow a)), s) empty_ta ((jThrow a), s)
 |  SynchronizedThrow1 : ∀ extTA a e s,
                      red extTA ((jsync (jThrow a) e), s) empty_ta
                          ((jThrow a), s)
 |  SynchronizedThrow2 : ∀ extTA a ad s,
                      red extTA ((jinsync (a) (jThrow ad)), s)
                          (ta_update_obs (ta_update_locks empty_ta Unlock a)
                                         (Synchronization a)) ((jThrow ad), s)
 |  SeqThrow : ∀ extTA a e2 s,
             red extTA ((Seq (jThrow a) e2), s) empty_ta ((jThrow a), s)
 |  CondThrow : ∀ extTA a e1 e2 s,
              red extTA ((Cond (jThrow a) e1 e2), s) empty_ta ((jThrow a), s)
 |  ThrowThrow : ∀ extTA a s,
               red extTA ((throw (jThrow a)), s) empty_ta ((jThrow a), s)
with  reds : (external_thread_action → J_thread_action) →
            list expr →
            (heap × locals) →
            J_thread_action →
            list expr →
            (heap × locals) → Prop :=
 |  ListRed1 : ∀ extTA e s ta e' s' es,
             red extTA (e, s) ta (e', s') →
             reds extTA (e :: es) s ta (e' :: es) s'
 |  ListRed2 : ∀ extTA es s ta es' s' v,
             reds extTA es s ta es' s' →
             reds extTA ((Val v) :: es) s ta ((Val v) :: es') s'.

Definition  red' := red (extTA2J).
Definition  reds' := reds (extTA2J).


Definition  jstep extTA :=
  clos_refl_trans_label (red extTA).

Definition  jstep' := jstep (extTA2J).

End Java.
```

A.6. VPJ: Auxiliary Definitions

```
Inductive  is_instance_type : jtype → Prop :=
 |  iit_void : is_instance_type Void
 |  iit_bool : is_instance_type Boolean
 |  iit_int : is_instance_type Integer
 |  iit_str : is_instance_type String
 |  iit_null : is_instance_type NullType
 |  iit_class : ∀ cn ps, is_instance_type (Class cn ps).

Inductive  is_fully_instantiated : jtype → Prop :=
 |  ift_bool : is_fully_instantiated Boolean
 |  ift_int : is_fully_instantiated Integer
 |  ift_str : is_fully_instantiated String
 |  ift_class : ∀ cn ps, params_fully_instantiated ps →
                         is_fully_instantiated (Class cn ps)
with  params_fully_instantiated : list (jvariance × jparam_type) → Prop :=
 |  pfi_nil : params_fully_instantiated nil
 |  pfi_cons : ∀ P Ps, is_fully_instantiated (RefType P) →
                       params_fully_instantiated Ps →
                       params_fully_instantiated ((JVInv, P) :: Ps).

Fixpoint  subst_jtvar_in_jparam (x : tvname) (y : jparam_type) (U : jparam_type)
                                {struct U} : jparam_type :=
match U with
 | TV tvn ⇒ if StringDec.eq_dec tvn x then y else TV tvn
 | TC C ⇒ TC (subst_jtvar_in_jclass x y C)
end
with  subst_jtvar_in_jclass (x : tvname) (y : jparam_type) (U : jclass_type)
                            {struct U} : jclass_type :=
match U with
 | TClass cn ps ⇒
     TClass cn (map (fun (p : jvariance × jparam_type) ⇒
                         let (v,t) := p in (v, (subst_jtvar_in_jparam x y t))) ps)
end.

Fixpoint  subst_jtvars_in_jparam (xs : list tvname) (ys : list jparam_type)
                                 (U : jparam_type) {struct U} : jparam_type :=
match U with
 | TV tvn ⇒
     fold_left (fun u xy ⇒ subst_jtvar_in_jparam (fst xy) (snd xy) u)
               (combine xs ys)
               (TV tvn)
 | TC (TClass cn ps) ⇒
     TC (TClass cn (map (fun (p : jvariance × jparam_type) ⇒
                             let (v,t) := p in
                             (v, (subst_jtvars_in_jparam xs ys t))) ps))
end.

Fixpoint  subst_jtvars_in_jclass (xs : list tvname) (ys : list jparam_type)
                                 (U : jclass_type) {struct U} : jclass_type :=
match U with
 | TClass cn ps ⇒
```

```
    TClass cn (map (fun (p : jvariance × jparam_type) ⇒
                        let (v,t) := p in (v, (subst_jtvars_in_jparam xs ys t))) ps)
end.

Definition  subst_jtvar_in_jtype (x : tvname) (y : jparam_type) (T : jtype) : jtype :=
match T with
 | RefType Tp ⇒ RefType (subst_jtvar_in_jparam x y Tp)
 | T' ⇒ T'
end.

Definition  subst_jtvars_in_jtype (xs : list tvname) (ys : list jparam_type)
                                  (T : jtype) : jtype :=
match T with
 | RefType Tp ⇒ RefType (subst_jtvars_in_jparam xs ys Tp)
 | T' ⇒ T'
end.

Fixpoint  subst_jtvar_in_expr (tvn : tvname) (t : jparam_type) (e : expr)
                              {struct e} : expr :=
match e with
 | New C args ⇒ New (subst_jtvar_in_jclass tvn t C)
                           (map (subst_jtvar_in_expr tvn t) args)
 | Cast T e ⇒ Cast (subst_jtvar_in_jtype tvn t T) e
 | Val v ⇒ Val v
 | BinOp e op e' ⇒ BinOp (subst_jtvar_in_expr tvn t e) op
                             (subst_jtvar_in_expr tvn t e')
 | Var v ⇒ Var v
 | LAss V e ⇒ LAss V (subst_jtvar_in_expr tvn t e)
 | FAcc e F D ⇒ FAcc (subst_jtvar_in_expr tvn t e) F D
 | FAss e F D e' ⇒ FAss (subst_jtvar_in_expr tvn t e) F D
                            (subst_jtvar_in_expr tvn t e')
 | Call e tvs m pns ⇒ Call (subst_jtvar_in_expr tvn t e)
                            (map (subst_jtvar_in_jparam tvn t) tvs) m
                            (map (subst_jtvar_in_expr tvn t) pns)
 | Block V T vo e ⇒ Block V (subst_jtvar_in_jtype tvn t T) vo
                             (subst_jtvar_in_expr tvn t e)
 | Sync V o' e ⇒ Sync V o' (subst_jtvar_in_expr tvn t e)
 | InSync V a e ⇒ InSync V a (subst_jtvar_in_expr tvn t e)
 | Seq e e' ⇒ Seq (subst_jtvar_in_expr tvn t e) (subst_jtvar_in_expr tvn t e')
 | Cond b e e' ⇒ Cond (subst_jtvar_in_expr tvn t b)
                            (subst_jtvar_in_expr tvn t e)
                            (subst_jtvar_in_expr tvn t e')
 | while b e ⇒ while (subst_jtvar_in_expr tvn t b) (subst_jtvar_in_expr tvn t e)
 | throw e ⇒ throw (subst_jtvar_in_expr tvn t e)
 | TryCatch e C v e' ⇒ TryCatch (subst_jtvar_in_expr tvn t e) C v
                                (subst_jtvar_in_expr tvn t e')
end.
Definition  subst_jtvars_in_expr (xs : list tvname) (ys : list jparam_type)
                                 (e : expr) : expr :=
  fold_left (fun e' xy ⇒ subst_jtvar_in_expr (fst xy) (snd xy) e')
            (combine xs ys) e.
```

A.7. Symbolic Library

A.7.1. AbstractBase

```
(* AbstractBase(String confname, String ctorname) *)
Definition  eAbstractBase : expr :=
  (* this.configName = confname; *)
     FAss jthis "configName" "AbstractBase" (Var "confname") ;
  (* this.ctorName = ctorname; *)
     FAss jthis "ctorName" "AbstractBase" (Var "ctorname") ;
  (* return this; *)
     jthis.
Definition  mAbstractBase : string × jmdef :=
  ("AbstractBase",
   (nil,
    ("confname", String) :: ("ctorname", String) :: nil,
    Class "AbstractBase" nil,
    eAbstractBase)).

Definition  ab_fields : fdecl :=
  fields_init (("configName", String) ::
               ("ctorName", String) :: nil).

Definition  ab_methods : mdecl := methods_init (mAbstractBase :: nil).
Definition  clAbstractBase : class :=
  (nil, CObject, ("configName" :: "ctorName" :: nil, ab_fields), ab_methods).
Definition  AbstractBase := ("AbstractBase", clAbstractBase).

Definition  CAbstractBase := TClass "AbstractBase" nil.
Definition  TCAbstractBase := TC CAbstractBase.
Definition  TAbstractBase := RefType TCAbstractBase.
```

A.7.2. AbstractGenerativeBase

```
(* AbstractGenerativeBase(String confname, String ctorname) *)
Definition  eAbstractGenerativeBase : expr :=
  (* super(confname, ctorname); *)
     Call jthis nil "AbstractBase" (Var "confname" :: Var "ctorname" :: nil) ;
  (* return this; *)
     jthis.
Definition  mAbstractGenerativeBase : string × jmdef :=
  ("AbstractGenerativeBase",
   (nil,
    ("confname", String) :: ("ctorname", String) :: nil,
    Class "AbstractGenerativeBase" nil,
    eAbstractGenerativeBase)).

Definition  agb_methods : mdecl :=
  methods_init (mAbstractGenerativeBase :: nil).
Definition  clAbstractGenerativeBase : class :=
  (nil, CAbstractBase, (nil, fields_empty), agb_methods).
Definition  AbstractGenerativeBase :=
```

```
  ("AbstractGenerativeBase", clAbstractGenerativeBase).

Definition  CAbstractGenerativeBase := TClass "AbstractGenerativeBase" nil.
Definition  TCAbstractGenerativeBase := TC CAbstractGenerativeBase.
Definition  TAbstractGenerativeBase := RefType TCAbstractGenerativeBase.
```

A.7.3. Semaphore

```
(* Semaphore(int value) *)
Definition  eSemaphore : expr :=
  (* this.val = value; *)
     FAss jthis "val" "Semaphore" (Var "value") ;
  (* return this; *)
     jthis.
Definition  mSemaphore : (string × jmdef) :=
  ("Semaphore",
   (nil,
    ("value", Integer) :: nil,
    Class "Semaphore" nil,
    eSemaphore)).

(* void acquire() *)
Definition  eacquire : expr :=
  (* boolean loop = true; *)
     Block "loop" Boolean (Some (Bool true)) (
      (* while (loop) ...  *)
         while (Var "loop") (
          (* synchronized (this) ...  *)
             jsync (jthis) (
              (* if (this.val > 0) ...  *)
                 :if ((FAcc jthis "val" "Semaphore") > (Val (Intg 0))) :then (
                  (* this.val = this.val - 1; *)
                     FAss jthis "val" "Semaphore"
                         (FAcc jthis "val" "Semaphore" - Val (Intg 1)) ;
                  (* loop = false; *)
                     "loop" ::= jfalse
                 ) :else (
                  (* this.wait(); *)
                     Call jthis nil wait (nil)
                 )
             )
         )
     ).
Definition  macquire : (string × jmdef) :=
  ("acquire",
   (nil,
    nil,
    Void,
    eacquire)).

(* void release() *)
Definition  erelease : expr :=
  (* synchronized (this) ...  *)
```

```
     jsync (jthis) (
      (* this.val = this.val + 1; *)
         FAss jthis "val" "Semaphore"
             (FAcc jthis "val" "Semaphore" + Val (Intg 1)) ;
      (* this.notifyAll(); *)
         Call jthis nil notifyAll (nil)
     ).
Definition  mrelease : (string × jmdef) :=
  ("release",
   (nil,
    nil,
    Void,
    erelease)).

Definition  sema_fields : fdecl := fields_init (("val", Integer) :: nil).

Definition  sema_methods : mdecl :=
  methods_init (mSemaphore :: macquire :: mrelease :: nil).
Definition  clSemaphore : jclass :=
  (nil, CObject, ("val" :: nil, sema_fields), sema_methods).
Definition  Semaphore :=
  ("Semaphore", clSemaphore).

Definition  CSemaphore := TClass "Semaphore" nil.
Definition  TCSemaphore := TC CSemaphore.
Definition  TSemaphore := RefType TCSemaphore.
```

A.7.4. AbstractChannel

```
(* AbstractChannel(String confname, String ctorname) *)
Definition  eAbstractChannel : expr :=
  (* super(confname, ctorname); *)
     Call jthis nil "AbstractGenerativeBase"
         (Var "confname" :: Var "ctorname" :: nil) ;
  (* this.msg = null; *)
     FAss jthis "msg" "AbstractChannel" jnull ;
  (* this.semaRecv = new Semaphore(0); *)
     FAss jthis "semaRecv" "AbstractChannel"
         (New (TClass "Semaphore" nil) (Val (Intg 0) :: nil)) ;
  (* this.semaReady = new Semaphore(0); *)
     FAss jthis "semaReady" "AbstractChannel"
         (New (TClass "Semaphore" nil) (Val (Intg 0) :: nil)) ;
  (* return this; *)
     jthis.
Definition  mAbstractChannel : string × jmdef :=
  ("AbstractChannel",
   (nil,
    ("confname", String) :: ("ctorname", String) :: nil,
    Class "AbstractChannel" ((JVInv, TV "T") :: nil),
    eAbstractChannel)).

(* public T receive() *)
Definition  ereceive : expr :=
```

```
  (* semaRecv.acquire(); *)
     Call (FAcc jthis "semaRecv" "AbstractChannel") nil "acquire" nil ;
  (* T ret; *)
     Block "ret" (TVar "T") None (
      (* ret = this.msg; *)
         "ret" ::= FAcc jthis "msg" "AbstractChannel" ;
      (* semaReady.release(); *)
         Call (FAcc jthis "semaReady" "AbstractChannel") nil "release" nil ;
      (* return ret; *)
         Var "ret").
Definition  mreceive : _ × jmdef :=
  ("receive",
   (nil,
    nil,
    TVar "T",
    ereceive)).

(* public void send(T msg) *)
Definition  esend : expr :=
  (* synchronized (this) ...  *)
     jsync (jthis) (
      (* this.msg = msg; *)
         FAss jthis "msg" "AbstractChannel" (Var "msg") ;
      (* semaRecv.release(); *)
         Call (FAcc jthis "semaRecv" "AbstractChannel") nil "release" nil ;
      (* semaReady.acquire(); *)
         Call (FAcc jthis "semaReady" "AbstractChannel") nil "acquire" nil ;
      (* this.msg = null; *)
         FAss jthis "msg" "AbstractChannel" jnull
     ).
Definition  msend : _ × jmdef :=
  ("send",
   (nil,
    ("msg", TVar "T") :: nil,
    Void,
    esend)).

Definition  ac_fields : fdecl :=
  fields_init (("msg", TVar "T") ::
               ("semaRecv", Class "Semaphore" nil) ::
               ("semaReady", Class "Semaphore" nil) :: nil).

Definition  ac_methods : mdecl :=
  methods_init (mAbstractChannel :: mreceive :: msend :: nil).
Definition  clAbstractChannel : class :=
  (("T", CAbstractBase) :: nil, CAbstractGenerativeBase,
   ("msg" :: "semaRecv" :: "semaReady" :: nil, ac_fields), ac_methods).
Definition  AbstractChannel :=
  ("AbstractChannel", clAbstractChannel).

Definition  CAbstractChannel T := TClass "AbstractChannel" (T :: nil).
Definition  TCAbstractChannel T := TC (CAbstractChannel T).
Definition  TAbstractChannel T := RefType (TCAbstractChannel T).
```

A.7.5. Elibrary, ELetProcess and EDestructor

```
(* class ELibrary extends Throwable *)
Definition  el_methods :=
  methods_init (("ELibrary", (nil, nil, Class "ELibrary" nil, jthis)) :: nil).
Definition  ELibrary : cname × class :=
  ("ELibrary",
   (nil, CThrowable, (nil, fields_empty), el_methods)).
Definition  CELibrary := TClass "ELibrary" nil.
Definition  TCELibrary := TC CELibrary.
Definition  TELibrary := RefType TCELibrary.

(* class ELetProcess extends ELibrary *)
Definition  elp_methods :=
  methods_init (("ELetProcess", (nil, nil, Class "ELetProcess" nil, jthis)) :: nil).
Definition  ELetProcess : cname × class :=
  ("ELetProcess",
   (nil, CELibrary, (nil, fields_empty), elp_methods)).
Definition  CELetProcess := TClass "ELetProcess" nil.
Definition  TCELetProcess := TC CELetProcess.
Definition  TELetProcess := RefType TCELetProcess.

(* class EDestructor extends ELibrary *)
Definition  ed_methods :=
  methods_init (("EDestructor", (nil, nil, Class "EDestructor" nil, jthis)) :: nil).
Definition  EDestructor : cname × class :=
  ("EDestructor",
   (nil, CELibrary, (nil, fields_empty), ed_methods)).
Definition  CEDestructor := TClass "EDestructor" nil.
Definition  TCEDestructor := TC CEDestructor.
Definition  TEDestructor := RefType TCEDestructor.
```

A.8. Global Expi Calculus: Auxiliary Definitions

```
Fixpoint  open_gproc_wrt_term_rec (k : nat) (t5 : term) (F : gproc) {struct F}:  gproc :=
match F with
 | gproc_out t u G ⇒ gproc_out (open_term_wrt_term_rec k t5 t)
                               (open_term_wrt_term_rec k t5 u)
                               (open_gproc_wrt_term_rec k t5 G)
 | gproc_in t G ⇒ gproc_in (open_term_wrt_term_rec k t5 t)
                           (open_gproc_wrt_term_rec (S k) t5 G)
 | gproc_bangin t G ⇒ gproc_bangin (open_term_wrt_term_rec k t5 t)
                                   (open_gproc_wrt_term_rec (S k) t5 G)
 | gproc_let g G H ⇒ gproc_let (open_dtor_wrt_term_rec k t5 g)
                               (open_gproc_wrt_term_rec (S k) t5 G)
                               (open_gproc_wrt_term_rec k t5 H)
 | gproc_gen T G ⇒ gproc_gen T (open_gproc_wrt_term_rec (S k) t5 G)
 | gproc_fork G H ⇒ gproc_fork (open_gproc_wrt_term_rec k t5 G)
                               (open_gproc_wrt_term_rec k t5 H)
 | gproc_null ⇒ gproc_null
end.

Fixpoint  open_gproc_wrt_nam_rec (k : nat) (nam5 : nam) (F : gproc) {struct F}:  gproc :=
match F with
 | gproc_out t u G ⇒ gproc_out (open_term_wrt_nam_rec k nam5 t)
                               (open_term_wrt_nam_rec k nam5 u)
                               (open_gproc_wrt_nam_rec k nam5 G)
 | gproc_in t G ⇒ gproc_in (open_term_wrt_nam_rec k nam5 t)
                           (open_gproc_wrt_nam_rec k nam5 G)
 | gproc_bangin t G ⇒ gproc_bangin (open_term_wrt_nam_rec k nam5 t)
                                   (open_gproc_wrt_nam_rec k nam5 G)
 | gproc_let g G H ⇒ gproc_let (open_dtor_wrt_nam_rec k nam5 g)
                               (open_gproc_wrt_nam_rec k nam5 G)
                               (open_gproc_wrt_nam_rec k nam5 H)
 | gproc_gen T G ⇒ gproc_gen T (open_gproc_wrt_nam_rec k nam5 G)
 | gproc_fork G H ⇒ gproc_fork (open_gproc_wrt_nam_rec k nam5 G)
                               (open_gproc_wrt_nam_rec k nam5 H)
 | gproc_null ⇒ gproc_null
end.

Definition  open_gproc_wrt_term t5 F5 := open_gproc_wrt_term_rec 0 F5 t5.

Definition  open_gproc_wrt_nam nam5 F5 := open_gproc_wrt_nam_rec 0 F5 nam5.

Inductive  lc_gproc : gproc → Prop :=
 |  lc_gproc_out : ∀ (t u : term) (G : gproc),
     (lc_term t) →
     (lc_term u) →
     (lc_gproc G) →
     (lc_gproc (gproc_out t u G))
 |  lc_gproc_in : ∀ (t : term) (G : gproc),
     (lc_term t) →
      ( ∀ x , lc_gproc ( open_gproc_wrt_term G (term_Var_f x) ) ) →
     (lc_gproc (gproc_in t G))
 |  lc_gproc_bangin : ∀ (t : term) (G : gproc),
```

```
     (lc_term t) →
      ( ∀ x , lc_gproc ( open_gproc_wrt_term G (term_Var_f x) ) ) →
     (lc_gproc (gproc_bangin t G))
 |  lc_gproc_let : ∀ (g : dtor) (G H : gproc),
     (lc_dtor g) →
      ( ∀ x , lc_gproc ( open_gproc_wrt_term G (term_Var_f x) ) ) →
     (lc_gproc H) →
     (lc_gproc (gproc_let g G H))
 |  lc_gproc_gen : ∀ (T : type) (G : gproc),
      ( ∀ x , lc_gproc ( open_gproc_wrt_term G (term_Var_f x) ) ) →
     (lc_gproc (gproc_gen T G))
 |  lc_gproc_fork : ∀ (G H : gproc),
     (lc_gproc G) →
     (lc_gproc H) →
     (lc_gproc (gproc_fork G H))
 |  lc_gproc_null :
     (lc_gproc gproc_null).

Fixpoint  fv_in_gproc (F5 : gproc) : vars :=
match F5 with
 | gproc_out t u G ⇒ (fv_in_term t) u (fv_in_term u) u (fv_in_gproc G)
 | gproc_in t G ⇒ (fv_in_term t) u (fv_in_gproc G)
 | gproc_bangin t G ⇒ (fv_in_term t) u (fv_in_gproc G)
 | gproc_let g G H ⇒ (fv_in_dtor g) u (fv_in_gproc G) u (fv_in_gproc H)
 | gproc_gen T G ⇒ (fv_in_gproc G)
 | gproc_fork G H ⇒ (fv_in_gproc G) u (fv_in_gproc H)
 | gproc_null ⇒ {}
end.

Fixpoint  fn_in_gproc (F5 : gproc) : vars :=
match F5 with
 | gproc_out t u G ⇒ (fn_in_term t) u (fn_in_term u) u (fn_in_gproc G)
 | gproc_in t G ⇒ (fn_in_term t) u (fn_in_gproc G)
 | gproc_bangin t G ⇒ (fn_in_term t) u (fn_in_gproc G)
 | gproc_let g G H ⇒ (fn_in_dtor g) u (fn_in_gproc G) u (fn_in_gproc H)
 | gproc_gen T G ⇒ (fn_in_gproc G)
 | gproc_fork G H ⇒ (fn_in_gproc G) u (fn_in_gproc H)
 | gproc_null ⇒ {}
end.

Fixpoint  subst_nam_in_gproc (nam5 : nam) (a : atom) (F : gproc) {struct F} : gproc :=
match F with
 | gproc_out t u G ⇒ gproc_out (subst_nam_in_term nam5 a t)
                               (subst_nam_in_term nam5 a u)
                               (subst_nam_in_gproc nam5 a G)
 | gproc_in t G ⇒ gproc_in (subst_nam_in_term nam5 a t) (subst_nam_in_gproc nam5 a G)
 | gproc_bangin t G ⇒ gproc_bangin (subst_nam_in_term nam5 a t)
                                   (subst_nam_in_gproc nam5 a G)
 | gproc_let g G H ⇒ gproc_let (subst_nam_in_dtor nam5 a g)
                               (subst_nam_in_gproc nam5 a G)
                               (subst_nam_in_gproc nam5 a H)
 | gproc_gen T G ⇒ gproc_gen T (subst_nam_in_gproc nam5 a G)
 | gproc_fork G H ⇒ gproc_fork (subst_nam_in_gproc nam5 a G)
```

```
                                (subst_nam_in_gproc nam5 a H)
 | gproc_null ⇒ gproc_null
end.

Fixpoint  subst_term_in_gproc (t5 : term) (x5 : atom) (F5 : gproc) {struct F5} : gproc :=
match F5 with
 | gproc_out t u G ⇒ gproc_out (subst_term_in_term t5 x5 t)
                                (subst_term_in_term t5 x5 u)
                                (subst_term_in_gproc t5 x5 G)
 | gproc_in t G ⇒ gproc_in (subst_term_in_term t5 x5 t) (subst_term_in_gproc t5 x5 G)
 | gproc_bangin t G ⇒ gproc_bangin (subst_term_in_term t5 x5 t)
                                    (subst_term_in_gproc t5 x5 G)
 | gproc_let g G H ⇒ gproc_let (subst_term_in_dtor t5 x5 g)
                                (subst_term_in_gproc t5 x5 G)
                                (subst_term_in_gproc t5 x5 H)
 | gproc_gen T G ⇒ gproc_gen T (subst_term_in_gproc t5 x5 G)
 | gproc_fork G H ⇒ gproc_fork (subst_term_in_gproc t5 x5 G)
                                (subst_term_in_gproc t5 x5 H)
 | gproc_null ⇒ gproc_null
end.

Fixpoint  subst_type_var_in_gproc (sub : list (string × type)) (F5 : gproc)
                                   {struct F5} : gproc :=
match F5 with
 | gproc_out t u G ⇒ gproc_out (subst_type_var_in_term sub t)
                                (subst_type_var_in_term sub u)
                                (subst_type_var_in_gproc sub G)
 | gproc_in t G ⇒ gproc_in (subst_type_var_in_term sub t)
                           (subst_type_var_in_gproc sub G)
 | gproc_bangin t G ⇒ gproc_bangin (subst_type_var_in_term sub t)
                                    (subst_type_var_in_gproc sub G)
 | gproc_let g G H ⇒ gproc_let (subst_type_var_in_dtor sub g)
                                (subst_type_var_in_gproc sub G)
                                (subst_type_var_in_gproc sub H)
 | gproc_gen T G ⇒ gproc_gen (subst_type_var_in_type sub T)
                            (subst_type_var_in_gproc sub G)
 | gproc_fork G H ⇒ gproc_fork (subst_type_var_in_gproc sub G)
                                (subst_type_var_in_gproc sub H)
 | gproc_null ⇒ gproc_null
end.

Fixpoint  subst_nam_with_term_in_gproc (t5 : term) (nam5 : nam) (F5 : gproc)
                                       {struct F5} : gproc :=
match F5 with
 | gproc_out t u G ⇒ gproc_out (subst_nam_with_term_in_term t5 nam5 t)
                                (subst_nam_with_term_in_term t5 nam5 u)
                                (subst_nam_with_term_in_gproc t5 nam5 G)
 | gproc_in t G ⇒ gproc_in (subst_nam_with_term_in_term t5 nam5 t)
                           (subst_nam_with_term_in_gproc t5 nam5 G)
 | gproc_bangin t G ⇒ gproc_bangin (subst_nam_with_term_in_term t5 nam5 t)
                                    (subst_nam_with_term_in_gproc t5 nam5 G)
 | gproc_let g G H ⇒ gproc_let (subst_nam_with_term_in_dtor t5 nam5 g)
                                (subst_nam_with_term_in_gproc t5 nam5 G)
```

```
                                (subst_nam_with_term_in_gproc t5 nam5 H)
 | gproc_gen T G ⇒ gproc_gen T (subst_nam_with_term_in_gproc t5 nam5 G)
 | gproc_fork G H ⇒ gproc_fork (subst_nam_with_term_in_gproc t5 nam5 G)
                                (subst_nam_with_term_in_gproc t5 nam5 H)
 | gproc_null ⇒ gproc_null
end.
```

A.9. Expi to Global Expi

```
Section ParametrizedByName.

Inductive translate1 : proc → gproc → Prop :=
| trans1_out : ∀ (t u : term) (P : proc) (G : gproc),
    lc_term t →
    lc_term u →
    translate1 P G →
    translate1 (proc_out t u P) (gproc_out t u G)
| trans1_in : ∀ (L : vars) (t : term) (P : proc) (G : gproc),
    lc_term t →
    (∀ x, x ∉ L →
       translate1 (open_proc_wrt_term P (term_Var_f x))
                  (open_gproc_wrt_term G (term_Var_f x))) →
    translate1 (proc_in t P) (gproc_in t G)
| trans1_bangin : ∀ (L : vars) (t : term) (P : proc) (G : gproc),
    lc_term t →
    (∀ x, x ∉ L →
       translate1 (open_proc_wrt_term P (term_Var_f x))
                  (open_gproc_wrt_term G (term_Var_f x))) →
    translate1 (proc_bangin t P) (gproc_bangin t G)
| trans1_let : ∀ (L : vars) (g : dtor) (P Q : proc) (G H : gproc),
    lc_dtor g →
    (∀ x, x ∉ L →
       translate1 (open_proc_wrt_term P (term_Var_f x))
                  (open_gproc_wrt_term G (term_Var_f x))) →
    translate1 Q H →
    translate1 (proc_let g P Q) (gproc_let g G H)
| trans1_new : ∀ (L : vars) (T : type) (P : proc) (G : gproc),
    (∀ x, x ∉ L →
       (∀ a, a ∉ L u {{ x }} →
          translate1 (subst_nam_with_term_in_proc (term_Var_f x) (Nam_f a)
                       (open_proc_wrt_nam P (Nam_f a)))
                     (open_gproc_wrt_term G (term_Var_f x)))) →
    translate1 (proc_new T P) (gproc_gen T G)
| trans1_fork : ∀ (P Q : proc) (G H : gproc),
    translate1 P G →
    translate1 Q H →
    translate1 (proc_fork P Q) (gproc_fork G H)
| trans1_null :
    translate1 proc_null gproc_null.

End ParametrizedByName.
```

A.10. Global Expi to VPJ: Auxiliary Definitions

```
Section ParametrizedGlobalPi.

Definition  declare_fields (fds : list vname) (Ts : list jtype) : fdecl :=
  fields_init (combine fds Ts).

Fixpoint  set_fields (cn : cname) (fds : list vname) (xs : list vname) : expr :=
match fds, xs with
 | fd :: fds', x :: xs' ⇒ FAss jthis fd cn (Var x) ; (set_fields cn fds' xs')
 | _, _ ⇒ NOP
end.

Fixpoint  localize_fields (cn : cname) (xs : list vname) (Ts : list jtype)
                          (e : expr) : expr :=
match (xs, Ts) with
 | (x :: ys, T :: Us) ⇒ Block x T None (x ::= FAcc jthis x cn ;
                                        localize_fields cn ys Us e)
 | (_, _) ⇒ e
end.

Fixpoint  arg_name (A : Type) (remains : list A) : string :=
match remains with
 | nil ⇒ ""
 | u :: us ⇒ String.String "a" (arg_name A us)
end.

Fixpoint  ctor_params (args : list jparam_type) : list (vname × jtype) :=
match args with
 | nil ⇒ nil
 | t :: ts ⇒ ((arg_name (t :: ts)), RefType t) :: (ctor_params ts)
end.

Fixpoint  ctor_pass_param_names (args : list jtype) : list expr :=
match args with
 | nil ⇒ nil
 | t :: ts ⇒ (Var (arg_name (t :: ts))) :: (ctor_pass_param_names ts)
end.
Fixpoint  ctor_pass_params (fn : string) (ctrs : list (string × list jtype))
                           : list expr :=
match ctrs with
 | nil ⇒ nil
 | (ctr, args) :: ctrs' ⇒
     (if (DecUtils.StringDec.eq_dec fn ctr) then
        ctor_pass_param_names args
      else
        map (fun x ⇒ jnull) args) ++ (ctor_pass_params fn ctrs')
end.

Fixpoint  ctor_args_names (fn : string) (args : list jtype)
                          : list (vname × jtype) :=
match args with
 | nil ⇒ nil
 | t :: ts ⇒ (String.append fn (arg_name (t :: ts)), t) :: (ctor_args_names fn ts)
```

```
end.

Fixpoint  ctor_args (ctrs : list (string × list jtype)) : list (vname × jtype) :=
match ctrs with
 | nil ⇒ nil
 | (fn, ts) :: ctrs' ⇒ (ctor_args_names fn ts) ++ (ctor_args ctrs')
end.

Fixpoint  init_ctor_args (obj : expr) (cn : cname) (fn : string)
                         (ts : list (@type Idents)) (e : expr) : expr :=
match ts with
 | nil ⇒ e
 | u :: us ⇒ (FAss obj (String.append fn (arg_name (u :: us))) cn
                   (Var (arg_name (u :: us)))) ; init_ctor_args obj cn fn us e
end.

Definition  fresh_cname (used : list cname) : cname :=
  fresh_str used.
Definition  fresh_mname (used : list mname) : mname :=
  fresh_str used.
Definition  fresh_vname (used : list vname) : vname :=
  fresh_str used.
Definition  fresh_tvname (used : list tvname) : tvname :=
  fresh_str used.
Fixpoint  fresh_tvname_list (n : nat) : list tvname :=
(fix fresh_tvname_list' (this : list tvname) (n : nat) : list tvname :=
 match n with
  | O ⇒ nil
  | S m ⇒ let tvn := (fresh_tvname this) in
            tvn :: (fresh_tvname_list' (tvn :: this) m)
 end) nil n.

Definition  fresh_atom (avoid : list atom) : atom :=
  proj1_sig (atom_fresh_for_list avoid).

Fixpoint  nat_to_vname (n : nat) : vname :=
match n with
 | O ⇒ "x"
 | S m ⇒ String.append "x" (nat_to_vname m)
end.

Fixpoint  ftype_for_type (c : @config Idents) (tn : Idents) (fn : Idents)
                         : list (string × list jtype) :=
match (f_type c "" fn) with
 | Some (ftype Xs Ts (type_Nested _ tn' _)) ⇒
     if (eq_Idents_dec c tn' tn) then
         (idtostr fn, map RefType (spitoj_type_list Ts)) :: nil
     else
         nil
 | _ ⇒ nil
end.

Fixpoint  ctors_for_type (c : @config Idents) (tn : Idents) (fns : list Idents)
                         : list (string × list jtype) :=
```

```
match fns with
 | nil ⇒ nil
 | fn :: fns' ⇒ (ftype_for_type c tn fn) ++ (ctors_for_type c tn fns')
end.

Fixpoint  code_match_ctor (c : @config Idents) (gn : Idents) (x : @term Idents)
                          (path : expr) {struct x} : expr :=
match x with
 | term_Ctor _ fn _ xs ⇒
     match (f_type c "" fn) with
      | Some (ftype _ _ (type_Nested _ tn _)) ⇒
          (fix and_matches (xs : list term) (e : expr) : expr :=
           match xs with
            | nil ⇒ e
            | y :: ys ⇒ and_matches ys ((code_match_ctor c gn y (FAcc path
                                (String.append (idtostr fn) (arg_name (y :: ys)))
                                (idtostr tn))) && e)
           end) xs
                (((FAcc (Cast TAbstractBase path) "configName" "AbstractBase")
                      == (Var "configName"))
                 && ((FAcc (Cast TAbstractBase path) "ctorName" "AbstractBase")
                           == Val (Str (idtostr fn))))
      | _ ⇒ jfalse
     end
 | _ ⇒ jtrue
end.

Fixpoint  code_match_ctors (c : @config Idents) (gn : Idents)
                           (xs : list (@term Idents)) {struct xs} : expr :=
match xs with
 | nil ⇒ jtrue
 | x :: ys ⇒ (code_match_ctor c gn x (Var (arg_name (x :: ys))))
              && (code_match_ctors c gn ys)
end.

Fixpoint  code_access_var (c : @config Idents) (gn : Idents) (x : @term Idents)
                          (T : @type Idents) (path : expr) {struct x}
                          : list (nat × expr × jtype) :=
match x with
 | term_Var_b n ⇒ (n, path, RefType (spitoj_type T)) :: nil
 | term_Ctor _ fn As xs ⇒
     match instantiate As (f_type c "" fn) with
      | Some (Ts, (type_Nested _ tn _)) ⇒
          (fix code_access_var_list (xs : list (@term Idents))
                                    (Ts : list (@type Idents)) {struct xs}
                                    : list (nat × expr × jtype) :=
           match xs, Ts with
            | x :: xs', T :: Ts' ⇒
                (code_access_var c gn x T (FAcc path
                     (String.append (idtostr fn) (arg_name (x :: xs')))
                     (idtostr tn))) ++ (code_access_var_list xs' Ts')
            | _, _ ⇒ nil
           end) xs Ts
```

```
      | _ ⇒ nil
     end
 | _ ⇒ nil
end.

Fixpoint  code_access_vars (c : @config Idents) (gn : Idents)
                           (xs : list (@term Idents)) (Ts : list (@type Idents))
                           : list (nat × expr × jtype) :=
match xs, Ts with
 | x :: ys, T :: Us ⇒ (code_access_var c gn x T (Var (arg_name (x :: ys))))
                        ++ (code_access_vars c gn ys Us)
 | _, _ ⇒ nil
end.

Fixpoint  project_n (n : nat) (xs : list (nat × expr × jtype))
                    : list (expr × jtype) :=
match xs with
 | nil ⇒ nil
 | (m, e, T) :: ys ⇒ if (NatDec.eq_dec n m) then
                           (e, T) :: (project_n n ys)
                      else
                           (project_n n ys)
end.

Definition  jnot (e : expr) : expr :=
  :if (e) :then jfalse :else jtrue.

Definition  code_return_term (c : @config Idents) (x : @term Idents)
                             (T : jparam_type) : expr :=
match (spitoj_term c (gamma_empty) x) with
 | Some (e, _) ⇒ e
 | _ ⇒ throw (New CEDestructor nil) ;
                 Cast (RefType T) jnull
end.

Fixpoint  code_compare_vars (v_1 : expr) (acclist : list (expr × jtype)) : expr :=
match acclist with
 | (path, RefType (TC _)) :: ys ⇒
     (Call v_1 nil "equals" (path :: nil)) && (code_compare_vars v_1 ys)
 | (path, RefType (TV _)) :: ys ⇒
     (v_1 == path) && (code_compare_vars v_1 ys)
 | _ ⇒ jtrue
end.

Inductive  has_symbolic_library : jprog → Prop :=
 |  hsl_rule : ∀ (J : jprog),
     J "AbstractBase" = Some (nil, CObject, ("configName" :: "ctorName" :: nil,
                                             ab_fields), ab_methods) →
     J "AbstractGenerativeBase" = Some (nil, CAbstractBase, (nil, fields_empty),
                                        agb_methods) →
     J "Semaphore" = Some (nil, CObject, ("val" :: nil, sema_fields),
                          sema_methods) →
     J "AbstractChannel" = Some (("T", CAbstractBase) :: nil,
                                 CAbstractGenerativeBase,
```

```
                                ("msg" :: "semaRecv" :: "semaReady" :: nil,
                                 ac_fields), ac_methods) →
    J "ELibrary" = Some (nil, CThrowable, (nil, fields_empty), el_methods) →
    J "ELetProcess" = Some (nil, CELibrary, (nil, fields_empty), elp_methods) →
    J "EDestructor" = Some (nil, CELibrary, (nil, fields_empty), ed_methods) →
    has_symbolic_library J.

End ParametrizedGlobalPi.
```