Team Red: The Sandstorm Theorem Prover

Deepak Garg Tom Murphy VII Greg Price
Jason Reed Noam Zeilberger

June 13, 2005

Abstract

We present the Sandstorm theorem prover, a forward-method focusing theorem prover for first-order intuitionistic logic.

1 Introduction

Focusing was a technique invented by Andreoli [1] to eliminate "don't care" non-determinism from proof search, and to carefully control "don't know" non-determinism. The forward method, on the other hand, blithely replaces both forms of non-determinism with exhaustive search, constructing proofs bottom up from initial axioms by applying all possible rules in a fair manner, and controlling combinatorial intractability by means of the subformula property. One of the simultaneously pleasing and painful features of the focusing forward-method for first-order intuitionistic logic is that it makes the logic's non-determinism very explicit. Rather than remaining hidden within the size of a large library of derived rules, non-determinism resurfaces in the potentially explosive number of ways to apply a much smaller number of derived focusing rules.

Building the Sandstorm theorem prover, we encountered this and other surprises and pitfalls – some apparently specific to the combination of focusing with the forward-method, others general to either technique, and some simply stemming from first-order logic. Focused derivations, for instance, provide an opportunity (which we simply note as future work) to build more compact proof terms, but the interaction of first-order unification with proof terms and derived rules is tricky.

The rest of the report describes the two major pieces of the theorem prover: the rule generation phase (Section 2), which mimics the focusing backward-method in order to construct derived rules for proving propositions, and rule application (Section 4), which exhaustively applies these rules, using a variety of optimizations and heuristics to provide good performance.

2 Rule generation

Rule generation begins with a proposition to prove, and constructs the set of derived rules that can possibly be applied to prove it. Since the forward prover is meant to reconstruct only focused backward derivations, we do not generate rules (and hence labels) for all subformulas, but rather each rule corresponds to a focus point – a synchronous formula occurring as an immediate subformula of an asynchronous connective. Correspondingly, each focus point is labeled, and the premises and conclusion of a rule refer only to labels and atomic formulas. This induces a natural representation of first-order formulas where the transitions between synchronous and asynchronous formulas are explicitly marked by a coercion. Figure 1 gives a BNF grammar for this representation. Note that every coercion from a synchronous to an asynchronous formula comes with a label, but that the reverse coercion does not have one. Before attempting to generate rules for a proposition, we explicitly convert it into this representation.

2.1 Derived inferences

Construction of rules proceeds from labeled formula to labeled formula or atom, with a synchronous phase followed by an asynchronous phase, and proceeds along lines similar to Andreoli's "Focussing interpretation" [1]. In the Focussing interpretation, a formula is mapped to a set of inferences for deriving it, where an inference is a set of premises and a conclusion. Our approach is essentially the same, but with a slight refinement – we define the mapping on entire sequents instead of formulas; for us this simplifies the interpretation of asynchronous connectives, but it seems it would not make a difference in the case of classical linear logic. Moreover, since inverting an asynchronous phase always leads to a single inference, and since the conclusion of all inferences for a sequent is the sequent itself, we leave both facts implicit. Thus we define two maps ϕ_a and Φ_s (which have both positive and negative versions), the first of which maps non-stable sequents to a set of premises needed to derive that sequent, and the latter of which sends stable sequents to *sets* of premise sets. Figure 2 gives the definitions of these functions for the propositional case, in terms of mostly elementary set operations (and Girard's so-called "locative tensor" [2]).

Some explanation of these definitions is necessary. The definitions

$$\begin{aligned} \phi_a^- \ (\Delta; \Omega, \mathcal{L} : S^- \vdash \gamma) &= \phi_a^- \ (\mathcal{L}, \Delta; \Omega \vdash \gamma) \\ \phi_a^+ \ (\Delta; \Omega \vdash \mathcal{L} : S^+) &= \phi_a^- \ (\Delta; \Omega \vdash \mathcal{L}) \end{aligned}$$

show how focusing ends, when encountering a synchronous formula in the asynchronous phase: the labels are pushed out to the sides, and the synchronous formula cannot be deconstructed further. The definitions of $\phi_a^- \ (\Delta; \Omega \vdash \gamma)$ and $\Phi_s^- \ (\Delta; S^- \vdash \gamma)$ are parametric in the right hand side γ. However, in the case of focusing on a negative atom, the right hand side is forced to be the same positive atom, and in that case there is one rule without premises for introducing the sequent. Note the difference between $\Phi_s^+ \ (\Delta; \cdot \vdash \top^+) = \{\emptyset\}$ and

$\Phi_s^+ (\Delta; \cdot \vdash \bot^+) = \emptyset$: focusing on $\bot$ on the right yields no rules, while focusing on $\top$ yields exactly one rule with no premises.

The first order case slightly complicates directly applying the functions defined in Figure 2 to rule generation. Decomposing a synchronous quantifier introduces a unification variable, while decomposing an asynchronous quantifier introduces a parameter. The set of parameters introduced by a rule must be kept track of, in order to ensure the condition that they do not escape into the rule's conclusion.

2.2 Proof terms, and interaction with unification

Adding proof term annotations to the rule generation procedure sketched provides further complications, but no significant hurdles. Our proof terms are standard, and we construct proof term combinators (as ML functions) for a derived rule simply by composing the combinators for each step. A more interesting approach might be to employ proof terms that reflect the structure of the focused derivation. For example, rather than deconstructing asynchronous connectives one at a time, they could be pattern matched against in a single step. The proof term

$$\lambda x.\mathbf{case}(x, y.\, \langle \mathbf{snd}\, y, \mathbf{fst}\, y \rangle\, , z.\, \langle \mathbf{snd}\, z, \mathbf{fst}\, z \rangle)$$

for $(A \wedge B) \vee (C \wedge D) \supset (B \wedge A) \vee (D \wedge C)$ would instead become the pattern

$$\begin{array}{rcl} \mathbf{inl}\, \langle y_1, y_2 \rangle & \Rightarrow & \langle y_2, y_1 \rangle \\ \mid \mathbf{inr}\, \langle z_1, z_2 \rangle & \Rightarrow & \langle z_2, z_1 \rangle \end{array}$$

In any case, our representation of proof terms shows deficiencies in the presence of unification. The combinators we construct operate on rigid terms, which cannot be unified. Yet, the proof terms for universal elimination ($M\ [t]$) and existential introduction ($[t, M]$) include terms from the quantification domain, and these terms must be discovered through unification. We solve this problem by applying the combinator to object level proof variables, and then performing a substitution. For example, the combinator for existential introduction takes a proof term M of type $A(\mathbf{X})$ and returns the proof term $[\mathbf{X}, M]$. Say that during application of the rule, the premise is matched against a proof term N of type $A(f(a))$. We cannot apply the combinator directly to N to produce the proof term $[\mathbf{X}, N]$, and then make the unification substitution $\mathbf{X} \mapsto f(a)$, because N is rigid. Instead, we apply the combinator to a fresh proof variable u to produce the proof term $[\mathbf{X}, u]$, make the unification substitution $\mathbf{X} \mapsto f(a)$, and finally the substitution $u \mapsto N$ to produce the proof term $[f(a), N]$. This trick works, but were we to rewrite the theorem prover, we would attempt to make the construction of proof terms account for unification more directly.

2.3 Stabilization and global assumptions

The theorem prover does not perform an initial stabilization phase. Rather, it annotates the proposition to be proven as a right asynchronous formula, and

then applies rule generation to that formula. This produces a single top-level rule for concluding the proposition (with sequents involving its immediate synchronous subformulas as premises), in addition to the rules for introducing all of the focus points and initial sequents. While this is partly only a cosmetic difference from stabilization, it has the disadvantage of making "global assumption elimination" more difficult to implement. Global assumption elimination takes advantage of the fact that after stabilization, hypotheses on the left of stable sequents will be propagated throughout rule generation for those sequents. Due to this invariant, occurrences of these hypotheses in premises and conclusions of rules can be omitted.

We foresee no great difficulty in modifying the theorem prover to perform initial stabilization and global assumption elimination. (Indeed, we have implemented a special case of it as optimizations on the generated rules, as explained in the next section.)

3 Middle-end

3.1 Rule Optimization

We perform a number of optimizations on the rules generated by the front-end.

Most of the dirtiness of what we do would go away if we had originally done stabilization of the goal sequent (as described in the previous section) rather than retrofit the solution that follows. If we had done stabilization, we would have no "top-level rule;" only a stabilization function that builds proof terms by invoking the theorem prover on each of its stable sequents. (These stable sequents correspond to the premises of our top-level rule).

What we did was treat the special case where the top-level rule has a single premise. In this case stabilization would produce a single invocation of the theorem prover, and all of the hypotheses of this premise would be globally known. In this scenario (which accounts for Prolog-style logic programs, i.e., the Horn fragment), we perform the following optimizations.

First, we remove the "top-level rule" from the rule set, because it only makes sense to apply it without weakening. Instead, we set as our goal the single premise of the rule, and only after finding a proof of that sequent do we apply the top rule in a separate phase.

Second, some parameters (as introduced by asynchronous $\forall$ and $\exists$ quantifiers) have scope that includes the entire proof search; they are global. Because these parameters are global, we rewrite them to new constants, which allows our rules to be more specific to the problem.[1] In the phase where we apply the top rule to create the final proof term, we must rewrite these global constants back into bound variables.

Third, we are able to do optimizations to our rules because we are in the scope of some *global assumptions*.

[1] There are several technical reasons why this is required for soundness if we do the next listed optimization, but they are difficult to explain here.

For each hypothesis $v : A$ of the premise of the top rule (our new goal sequent), we know that it will be known (in scope) everywhere within the proof. Therefore, we find every occurrence of it in a rule (using syntactic equality, conservatively) and remove it. There are four ways that the formula A can appear within a rule.

1. $$\frac{P_1 \quad \ldots \quad P_n}{x : A, \Gamma \Longrightarrow C}$$

   Here we're introducing the formula in the conclusion. We should remove it from the hypothesis set, and then need to patch up the proof term generated by the rule.

   $$x : A, \Gamma \Longrightarrow M : C$$

   becomes

   $$\Gamma \Longrightarrow \texttt{let x = v in M end} : C$$

2. $$\frac{P_1 \quad \ldots \quad P_n}{\Gamma \Longrightarrow A}$$

   The rule is removed. There is no point in concluding A, since it can never be used (due to case 4). This includes initial sequents.

3. $$\frac{x : A, \Gamma \Longrightarrow C_1 \quad \ldots \quad P_n}{\Gamma \Longrightarrow C}$$

   Because A is globally known, we remove it from the premise, leaving just $\Gamma \Longrightarrow C_1$. The proof term generator expects a proof of $x : t, \Gamma \Longrightarrow C_1$, but we'll give it a $\Gamma \Longrightarrow C_1$. This is justified by weakening, so we need not modify the proof terms.

4. $$\frac{\Gamma_1 \Longrightarrow A \quad \ldots \quad P_n}{\Gamma \Longrightarrow C}$$

   Here, because t is already known, requiring it as a premise is unnecessary. We remove the first premise. The proof term generator for the rule (which is encoded as a function) expects n proofs but we will give it $n-1$. Therefore we must wrap the function to pass along the global variable for t (v, from above) as the proof of the deleted premise.

Additionally, the new goal sequent

$$v : A, \Gamma \Longrightarrow C$$

becomes

$$\Gamma \Longrightarrow C.$$

The proof term generator (for the top rule) expects that the proof it receives relies on $v : A$. Indeed it does; however, we remove $v : A$ from the hypotheses because, as far as proof search is concerned, these variables are "global."[2]

Because we only attempted this optimization late (after realizing that our prover worked quite badly on "logic program"-style propositions), and discovered several bugs and unexpected requirements while implementing it, we actually only implement item 1. The remainder should be quite easy to implement now.

4 Back-end

The back-end of the theorem prover manages the actual process of applying rules in order to attempt to prove a goal. We use the same general structure as the *Otter Loop* presented in the course notes.

In order to implement this loop we have the following pieces:

- A representation of facts (which are things we know)
- The set(s) of facts (active and passive sets; called the *database*)
- A way to select a *given* fact
- A way to generate all possible applications of rules using the given fact
- A way of generating all possible contractions of a fact
- A way of checking if a fact that we learn is already known (*subsumed* by the database)
- A way to insert new facts into the database.

4.1 Unification

Due to our experience with implementing unification for programming languages (where back-tracking during typechecking is rare), we decided to use an imperative implementation of unification.

What we do is to have two representations of terms: *frozen* and *thawed*. Frozen terms are purely symbolic. Thawed terms use `thawed_term option ref` to represent variables, so that we can unify them imperatively. By *thawing* a set of frozen terms together (by way of an imperative *thaw context*, which represents the typically invisible scope of the quantification over existential variables), we can get thawed terms that are ready for unification.

The idea is to thaw terms only immediately before unification (which is more efficient because of its use of references), and then to *freeze* them back

[2]Note that we are then left with a goal of $\Longrightarrow L_n$ for some n. If only one rule concludes L_n, and it itself also has a single premise (if such a thing is even possible!), then we might even consider applying this procedure recursively.

into purely symbolic terms (again by way of a *freeze context*). Unfortunately, due to a set of unfortunate design decisions and requirements of the problem, it turns out that we often need to treat thawed terms themselves functionally.[3] Therefore, we have a way of treating reference cells in thawed rules as themselves variables. Thus we have to *copy* thawed rules (which is analogous to thawing), and perform unification on these thawed copies. The backtracking inherent in theorem proving, coupled with the more local scope of the unification problems involved makes imperative unification less attractive than it is for type checking a programming language.

Although the code is verbose and probably less efficient than a purely functional version, there is some benefit to the representation that we stumbled upon. Because we have different types of frozen and thawed terms (which could be the same in a functional implementation) and different types of freeze contexts, thaw contexts, and several different types of substitutions (which may have all been the same substitution type in a functional implementation), we were able to use the SML type system to prevent us from making many bugs. (Indeed, despite this code being painful and difficult to write, we had no difficult bugs within it.) Of course, the same kind of type discipline could be used in a functional program (perhaps with no cost through the use of phantom type parameters), but that would require even more foresight than we apparently possess.

4.2 Universal Term Representation

As it turns out, after rule generation we don't care about propositions any more. We're really left with the task of looking at a sequent, and seeing if it *matches* a premise of a rule. To match conclusions, for instance, they must conclude the same label with the same arguments under some unifying substitution. Here we make the observation that this is really a kind unification: if we treat labels as function symbols, then we can use our existing unification algorithm to do this matching. To be specific, we rewrite each label as follows:

$$\mathcal{L}_i(t_1, \ldots, t_n) \;\mapsto\; \mathsf{U}(L_i(t_1, \ldots, t_n))$$

... where U is a single unary universal atomic proposition, and L_i is a distinct function symbol always used to translate $\mathcal{L}_i$. Because *all* propositions are now represented as U applied to a single term, we can avoid mentioning U and simply represent propositions as (*translated*) terms.

This representation has several benefits. First, we avoid having to write two "layers" of code: one that treats propositions and one that treats their arguments. Second, we are able to represent parametric conclusions cleanly. A "left" rule, such as:

$$\frac{x{:}\mathcal{L}_1, y{:}\mathcal{L}_2 \Longrightarrow R}{z{:}\mathcal{L}_3 \Longrightarrow R}$$

[3] For instance, our frozen term datatype cannot represent restrictions on parameter dependencies.

... is just written as:

$$\frac{x{:}\mathsf{U}(L_1()), y{:}\mathsf{U}(L_2()) \Longrightarrow \mathsf{U}(\mathbf{X})}{z{:}\mathsf{U}(L_3()) \Longrightarrow \mathsf{U}(\mathbf{X})}$$

... where $\mathbf{X}$ is an existential variable.

We can also represent empty conclusions in sequents (that arise from $\bot$-elimination) in the same way.

As a result, the type of sequents throughout the backend is simply `(var` $\times$ `term) list` $\times$ `term`. A fact is just a sequent paired with a proof term of that sequent.

4.3 Database

Our database consists logically of a *passive* and *active* set. The passive set is those facts that we know to be true but which have not been *activated*—only activated rules are eligible for rule application.

We actually represent both of these sets within the same data structure. This structure (a discrimination tree) is optimized for asking the following question:

- Given a term t, return all passive facts that contain a generalization of t.

In addition, we also support the following operations (less efficiently):

- Given a term t, return all candidate active facts f such that f contains a term t' that *might* unify with t.
- Return all facts in the database.

By fact f contains term t, we mean either that the conclusion of the fact is the term, or it is one of the hypotheses.

The implementation details are interesting, but perhaps beyond the scope of this writeup.

4.4 Selection of Given Fact

Our selection of the given fact is highly simplistic. At a ratio of 5:1, we select "promising" sequents to "eldest sequents." A promising sequent is simply one whose conclusion is a candidate unifier for the theorem prover's goal.

4.5 Rule Application

In the general case there are many, *many* ways that a rule can apply. However we have two sorts of restrictions that help us to narrow down this explosion of possibilities.

One is that the given sequent *must* be involved as the match for at least one of the premises of the rule. This is standard for the Otter loop.

The second (which, for all we know, is also standard) is that every premise has at least one rigid term, where a rigid term is either a non-general conclusion[4], or some hypothesis. This allows us to use our discrimination tree data structure to find all candidate sequents that might unify with the premise at all (by searching for the rigid term).

Even given a matching of sets of candidate facts to premises, we still have many, *many* ways of applying the rule. For instance, we can instantiate the premise with the fact in each of the listed ways.

$$\begin{array}{lrcl} \text{fact} & z : f(a), w : f(b), u : g(c) & \Longrightarrow & h() \\ \text{premise} & \Gamma, x : f(X), y : g(Y) & \Longrightarrow & h() \end{array}$$

1. $X \leftarrow a$, $Y \leftarrow c$, $\Gamma = w : f(b)$
2. $X \leftarrow b$, $Y \leftarrow c$, $\Gamma = z : f(a)$
3. $Y \leftarrow c$, $\Gamma = z : f(a), w : f(b)$
4. $X \leftarrow a$, $\Gamma = w : f(b), u : g(c)$
5. $X \leftarrow b$, $\Gamma = z : f(a), u : g(c)$
6. $\Gamma = z : f(a), w : f(b), u : g(c)$ (ok since we match the rigid conclusion)

One of the reasons for so many choices is that we may have several choices of hypothesis to unify with any hypothesis in the premise (as seen in the difference between 1 and 2). Another reason is that we may discard a hypothesis in the premise by weakening the fact (as seen in the difference between 1 and 4).

In our code we are essentially just solving this huge combinatorial problem by brute force, except that we are able to give up early on many paths (we hope) by doing unification eagerly.

4.6 Contraction

Each application of a rule gives us a fact. Unlike the propositional case, we cannot apply contraction to this fact eagerly, because contractions that unify may leave us with a fact that is less general than before. Furthermore, there may be multiple possible contractants of each fact. Therefore, we must insert the original fact into the database, and also each (incomparable) fact in its "contraction closure" into the database as well.

However, before beginning the process we can do a simple form of eager contraction. Any two hypotheses that are syntactically identical (other than their variable, of course) can be contracted eagerly with no loss of information. (We can always recover the prior fact by weakening, if we like.) Implementing this simple pre-pass resulted in a prover that was 200 times faster!

We compute the contraction closure in the following way. Suppose we have a sequent:

$$x_0 : t_0, \ldots, x_n : t_n \Longrightarrow t_c$$

[4]General conclusions take the form $\mathsf{U}(\mathbf{X})$

We logically produce every possible partitioning of the set $\{x_0 \ldots x_n\}$[5] Each partitioning is in the contraction closure iff each partition in the partitioning can be unified. As we do this we must account for the fact that multiple variables in the hypotheses are collapsed into one by rewriting the proof term.

This method is correct but produces many partitionings that will not be in the closure. Some of these doomed partitionings share prefixes of partitions $P_1, \ldots, P_m$ where some $P_i (i \leq m)$ is not unifiable; in this case we can benefit from realizing that *any* partitioning with this prefix is doomed, and stop early. The way we accomplish this is by unifying eagerly as we generate partitionings recursively.

4.7 Subsumption Checking

After computing the closure, we want to insert each of the facts from the closure into the database. However, some of these facts may be in the database already—or worse—may be less useful than facts that are already in the database. We don't want to insert these; the check of whether a fact should be added or not is *subsumption* checking (a fact is *subsumed* by another fact if it is less general than it).

Discrimination trees are designed to make the subsumption check efficient. Our optimized question (*is there a generalization of t in the database?*) allows us to find sequents that have more general conclusions and more general hypotheses; we then use this approximate information to ensure that an actual matching exists. (As in the rule application case, we must consider multiple ways of matching hypotheses in the two facts, as well as weakenings in one direction.) If any fact in the database (active *or* passive set) subsumes our candidate insertee, then it is not inserted.

If a fact is not subsumed, it is inserted in the passive set with the current epoch, extending the discrimination tree with the appropriate branches.

References

[1] Jean-Marc Andreoli. Focussing and proof construction. *Annals of Pure and Applied Logic*, 107(1):131–163, 2001.

[2] Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[5] A partitioning of a set is a set of disjoint and exhaustive subsets of the original set. Each set is a partition.

S, T	synchronous formulas
A, B	asynchronous formulas
t	terms
$\mathcal{L}$	labels

$$
\begin{array}{rcl}
S^-, T^-, \ldots & ::= & P^-(t_1, \ldots, t_n) \\
& | & S^- \wedge T^- \\
& | & \top^- \\
& | & S^+ \supset T^- \\
& | & \forall x.S^- \\
& | & A^-
\end{array}
$$

$$
\begin{array}{rcl}
A^-, B^-, \ldots & ::= & P^-(t_1, \ldots, t_n) \\
& | & A^- \wedge B^- \\
& | & \top^- \\
& | & A^- \vee B^- \\
& | & \bot^- \\
& | & \exists x.A^- \\
& | & \mathcal{L} : S^-
\end{array}
$$

$$
\begin{array}{rcl}
S^+, T^+, \ldots & ::= & S^+ \wedge T^+ \\
& | & \top^+ \\
& | & S^+ \vee T^+ \\
& | & \bot^+ \\
& | & \exists x.S^+ \\
& | & A^+
\end{array}
$$

$$
\begin{array}{rcl}
A^+, B^+, \ldots & ::= & P^+(t_1, \ldots, t_n) \\
& | & A^+ \wedge B^+ \\
& | & \top^+ \\
& | & A^- \supset B^+ \\
& | & \forall x.A^+ \\
& | & \mathcal{L} : S^+
\end{array}
$$

Figure 1: Annotated formulas

$$
\begin{array}{rcl}
\phi_a^-\ (\Delta; \cdot \vdash \gamma) & = & \{\Delta \vdash \gamma\} \\
\phi_a^-\ (\Delta; \Omega, A^- \wedge B^- \vdash \gamma) & = & \phi_a^-\ (\Delta; \Omega, A^-, B^- \vdash \gamma) \\
\phi_a^-\ (\Delta; \Omega, A^- \vee B^- \vdash \gamma) & = & \phi_a^-\ (\Delta; \Omega, A^- \vdash \gamma) \cup \phi_a^-\ (\Delta; \Omega, B^- \vdash \gamma) \\
\phi_a^-\ (\Delta; \Omega, \top^- \vdash \gamma) & = & \phi_a^-\ (\Delta; \Omega \vdash \gamma) \\
\phi_a^-\ (\Delta; \Omega, \bot^- \vdash \gamma) & = & \emptyset \\
\phi_a^-\ (\Delta; \Omega, P^-(t_1, \ldots, t_n) \vdash \gamma) & = & \phi_a^-\ (P^-(t_1, \ldots, t_n), \Delta; \Omega \vdash \gamma) \\
\phi_a^-\ (\Delta; \Omega, \mathcal{L} : S^- \vdash \gamma) & = & \phi_a^-\ (\mathcal{L}, \Delta; \Omega \vdash \gamma) \\
 & & \\
\phi_a^+\ (\Delta; \Omega \vdash A^+ \wedge B^+) & = & \phi_a^+\ (\Delta; \Omega \vdash A^+) \cup \phi_a^+\ (\Delta; \Omega \vdash B^+) \\
\phi_a^+\ (\Delta; \Omega \vdash \top^+) & = & \emptyset \\
\phi_a^+\ (\Delta; \Omega \vdash A^- \supset B^+) & = & \phi_a^+\ (\Delta; \Omega, A^- \vdash B^+) \\
\phi_a^+\ (\Delta; \Omega \vdash P^+(t_1, \ldots, t_n)) & = & \phi_a^-\ (\Delta; \Omega \vdash P^+(t_1, \ldots, t_n)) \\
\phi_a^+\ (\Delta; \Omega \vdash \mathcal{L} : S^+) & = & \phi_a^-\ (\Delta; \Omega \vdash \mathcal{L}) \\
 & & \\
\Phi_s^-\ (\Delta; P^-(t_1, \ldots, t_n) \vdash P^+(t_1, \ldots, t_n)) & = & \{\emptyset\} \\
\Phi_s^-\ (\Delta; P^-(t_1, \ldots, t_n) \vdash \gamma) & = & \emptyset \qquad \text{if } \gamma \neq P(t_1, \ldots, t_n) \\
\Phi_s^-\ (\Delta; S^- \wedge T^- \vdash \gamma) & = & \Phi_s^-\ (\Delta; S^- \vdash \gamma) \cup \Phi_s^-\ (\Delta; T^- \vdash \gamma) \\
\Phi_s^-\ (\Delta; \top^- \vdash \gamma) & = & \emptyset \\
\Phi_s^-\ (\Delta; S^+ \supset T^- \vdash \gamma) & = & \Phi_s^+\ (\Delta; \cdot \vdash S^+) \bowtie \Phi_s^-\ (\Delta; T^- \vdash \gamma) \\
\Phi_s^-\ (\Delta; A^- \vdash \gamma) & = & \left\{\phi_a^-\ (\cdot; A^- \vdash \gamma)\right\} \\
 & & \\
\Phi_s^+\ (\Delta; \cdot \vdash S^+ \wedge T^+) & = & \Phi_s^+\ (\Delta; \cdot \vdash S^+) \bowtie \Phi_s^+\ (\Delta; \cdot \vdash T^+) \\
\Phi_s^+\ (\Delta; \cdot \vdash \top^+) & = & \{\emptyset\} \\
\Phi_s^+\ (\Delta; \cdot \vdash S^+ \vee T^+) & = & \Phi_s^+\ (\Delta; \cdot \vdash S^+) \cup \Phi_s^+\ (\Delta; \cdot \vdash T^+) \\
\Phi_s^+\ (\Delta; \cdot \vdash \bot^+) & = & \emptyset \\
\Phi_s^+\ (\Delta; \cdot \vdash A^+) & = & \left\{\phi_a^+\ (\Delta; \cdot \vdash A^+)\right\}
\end{array}
$$

Note: $\mathbb{C} \bowtie \mathbb{D} = \{c \cup d \mid c \in \mathbb{C}, d \in \mathbb{D}\}$ is Girard's "locative tensor"

Figure 2: Definitions of ϕ and Φ for propositional fragment