Distributivity in Incompletely Specified Type Hierarchies: Theory and Computational Complexity [†]

Stephen J. Hegner [‡]
University of Bergen
Department of Informatics
Bergen High-Technology Center
N-5020 Bergen, Norway

Internet: hegner@ii.uib.no

(Revised: August 1994)

Keywords: knowledge representation, type hierarchies, distributivity, feature structures, computational complexity.

[†]This work was supported by a grant from the Norwegian Research Council.

[‡]Current address: Department of Computer Science and Electrical Engineering, Votey Building, University of Vermont, Burlington, VT 05405, USA; Internet: hegner@emba.uvm.edu.

Abstract

Type hierarchies are a central feature of current systems which employ feature structures for the representation and manipulation of linguistic knowledge, such as TFS, ALE, CUF, and LKB. The system CUF stands out amongst these in its very general language for describing such hierarchies. While the other systems use a description language which allows only the specification of meets (greatest lower bounds of types), CUF provides a very general type definition language which allows a full complement of Boolean operations. This flexibility permits much simpler and more natural specifications of certain types of knowledge representation which occur naturally in paradigms such as HPSG. However, along with this increased power of expressiveness comes an increase in computational complexity. Specifically, while the simpler type systems are in fact so restrictive that any admitted specification is semantically meaningful, the power of the CUF type-specification language results in highly nontrivial decision question about well-formedness. These well-formedness questions revolve around the property of distributivity, as any semantically meaningful hierarchy — that is, one in which meet represents intersection of object classes and join represents union — must be distributive.

The contribution of this report is twofold. First of all, we provide a very general development of extension problems for partial type hierarchies to full distributive hierarchies, which is exactly the problem one needs to solve to determine well-formedness and other properties of type hierarchies in the CUF spirit. Secondly, we analyze the computational complexity of deciding the existence of various forms of distributive extensions. We show that under relatively weak assumptions concerning the height and fanout of the partial specification, all of the various decision problems are NP-complete. However, in the process of developing the formalism necessary for this detailed analysis, we also identify some close connections to certain kinds of satisfiability problems in propositional logic, which will hopefully prove useful in identifying useful special cases in which the computational problems are tractable.

0. Introduction

0.1 Motivation

The use of unification-based formalisms in computational linguistics has become commonplace. Systems based upon such formalisms depend fundamentally upon logical inference, in which constraints are collected as the parsing process proceeds, and the final results are the models (often taken to be feature structures) of the constraints. There are basically two flavors of such formalisms. In one school, which is currently represented most visibly by LFG [Kap89], the parsing process has a context-free backbone, and the unification-based formalism is used as a constraint-enforcement mechanism to "weed out" incorrect parses. Both Johnson's [Joh88] and Shieber's books [Shi92] describe recent theoretical efforts in this direction. In the other school, which is currently represented most visibly by HPSG [PS87], the context-free backbone is eschewed entirely. Rather, the entire parsing process becomes one of constraint satisfaction. This is a rather intriguing idea, but it also presents several new and unique problems which must be addressed if it is to be successful. One aspect is that "ordinary" feature structures, as described, for example in [Shi86], are no longer adequate to describe the inference process. Rather, so-called *typed* feature structures are used to express certain kinds of constraints. In typed feature structures, there is an extant finite type hierarchy, and each node of the feature structure is assigned a type. These types become an integral part of the inference mechanism.

In response to the idea of deductive computation in the context of typed feature structures, a number of specialized programming languages have evolved. Among the most visible are TFS [Zaj91], [Zaj92], ALE [Car92a], and CUF [DD93]. Recently, a similar system, LKB, has arisen from the ACQUILEX project [BdPC93]. In addition, the programming language LIFE [AP93], [Ait93] and its predecessor LOGIN [AN86], although not specifically designed for linguistic knowledge representation and manipulation, nonetheless shares many ideas with these linguistically motivated languages. No two of these systems handle the notion of typing in exactly the same way. However, they all begin with a simple, parameterless, type hierarchy. That is to say, a hierarchy in which we can say things such "Type a is subtype of type b" ($a \leq b$), or "An object is of type c if and only if it is of type a and of type b" ($c = a \wedge b$).

Despite their individual differences with respect to the declaration of the type hierarchy, these systems may be partitioned into two broad categories. On the one hand, TFS, ALE, LKB, LOGIN, and LIFE all build the type hierarchy in a particular way from a specification which is a partially ordered set. CUF, on the other hand, works with a much more powerful language, and therefore admits the specification of more general hierarchies. (See [Man93] for a comparison of CUF to the systems ALE and TFS.) With this generality, however, come some more difficult computational problems as well. Indeed, in other contexts, the tradeoff between expressive power and computational complexity in type hierarchies is well-known [LB87]. In this report, we study in detail some of the special algebraic aspects, and the associated computational complexity problems, of type declarations of the form admitted by CUF. Specifically, we address the question of when such a set of declarations is semantically well-formed. We shall argue that such hierarchies should be distributive, and we shall then show that the computational complexity of testing whether or not such a hierarchy is indeed distributive is NP-complete, even under relatively constrained circumstances.

In so doing, we shall not be concerned with the underlying logical language at all in this report. We take this tact for two reasons. First of all, having the type hierarchy be "satisfactory" from an algebraic point of view, while not a sufficient condition for the system to be well-specified overall, is certainly a necessary one. In particular, our work relates directly to an important subproblem which the CUF system must solve. Secondly, it is our thesis that a type/sort system which yields full Turing-complete power is not the optimal way to approach HPSG-style parsing, because in so doing we give up any hope of having automatically decidable parsing, such as off-line parsability in LFG provides [Joh88, 3.5]. In future reports, we shall document this idea of decidable parsing in HPSG-style contexts more fully.

We will not provide a self-contained introduction to typed feature structures here. In fact, we will not explicitly discuss features structures at all. Rather, we will focus completely upon the algebraic properties of the form of type hierarchies which appear in CUF and ALE, and at least implicitly, in TFS and LIFE as well. For the interested reader, Carpenter's book [Car92b] provides a rather thorough introduction to the ideas of typed feature structures, and how type hierarchies are involved in their definition[1]. However, virtually all of this report, with the exception of some details in motivating examples, may be understood without any knowledge of feature structures.

Within this introductory section, we assume a minimal acquaintance with the algebraic notions of partial order and lattice. We also present very simple examples of CUF and ALE syntax, although no detailed knowledge of these systems is needed to understand what is written here.

0.2 A Simple Technique for Realizing Distributive Type Hierarchies

Before describing the type declaration system employed by CUF, it is helpful to understand the features and limitations of the simpler method employed by the other systems mentioned above. Let us start by considering the type hierarchy depicted in Figure 0.1, which is taken from [AN86]. This particularly hierarchy is a lattice, and as per the usual convention for a pictorial representation of a lattice, one follows the arrows upwards to a common point to get the join of two elements, and downwards to get the meet.

An intuitive idea with any such hierarchy is that each type represents a subset of the objects in some universe $\mathcal{U}$. For any type t, let us write $\mathcal{V}(t)$ to denote

[1] Be aware that Carpenter expresses type hierarchies "upside down" from the way that we do. $\top$ denotes the empty type and $\bot$ the universal type in his formalism.

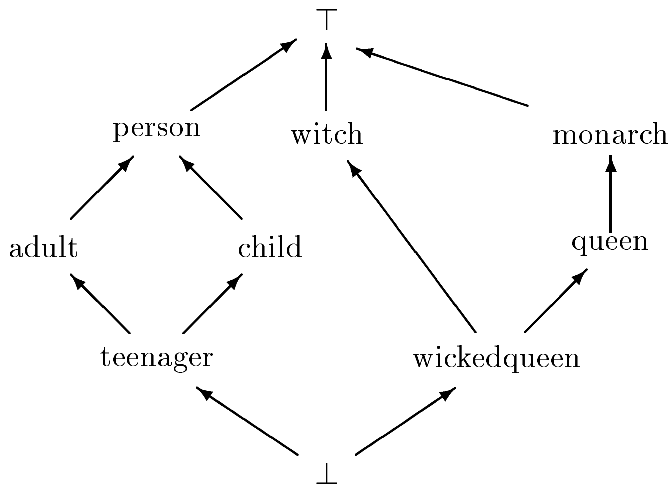

Figure 0.1: A non-distributive type hierarchy.

the set of all objects of type t. The symbol $\top$ denotes the universal type, and so $\mathcal{V}(\top) = \mathcal{U}$, and $\perp$ denotes the empty type, so that $\mathcal{V}(\perp) = \emptyset$. Also, as a first assumption, let us take the lattice operations to represent the natural operations on sets of objects. Thus, join represents union, and meet represents intersection, of these sets of instances. For example, we have that $\mathcal{V}(\text{adult}) \cup \mathcal{V}(\text{child}) = \mathcal{V}(\text{person})$ and $\mathcal{V}(\text{adult}) \cap \mathcal{V}(\text{child}) = \mathcal{V}(\text{teenager})$. However, we must then have that $\mathcal{V}(\text{monarch}) \cap \mathcal{V}(\text{witch}) = \mathcal{V}(\text{wickedqueen})$ and $\mathcal{V}(\text{queen}) \cap \mathcal{V}(\text{witch}) = \mathcal{V}(\text{wickedqueen})$. Thus, $\mathcal{V}(\text{monarch}) \cap \mathcal{V}(\text{witch}) = \mathcal{V}(\text{queen}) \cap \mathcal{V}(\text{witch})$. But, arguing similarly with union (*qua* join), we can show that $\mathcal{V}(\text{monarch}) \cup \mathcal{V}(\text{witch}) = \mathcal{V}(\text{queen}) \cup \mathcal{V}(\text{witch})$. However, $\mathcal{V}(\text{monarch}) \cap \mathcal{V}(\text{witch}) = \mathcal{V}(\text{queen}) \cap \mathcal{V}(\text{witch})$ means that the monarchs that are witches are precisely the queens that are witches, while $\mathcal{V}(\text{monarch}) \cup \mathcal{V}(\text{witch}) = \mathcal{V}(\text{queen}) \cup \mathcal{V}(\text{witch})$ means that the monarchs that are not witches are precisely the queens that are not witches. In other words, we must have that $\mathcal{V}(\text{monarch}) = \mathcal{V}(\text{queen})$. Thus, if we require that each type be distinct, and if we require that the operations of join and meet correspond to union and intersection of the corresponding sets of objects in the universe, then we cannot admit such a lattice as the definition of our type hierarchy. What we need is precisely that the lattice be distributive[2].

The "$\top$-witch-monarch-queen-wickedqueen" component of this lattice is a particular instance of what is known as a *pentagon*; a generic pentagon is depicted in Figure 0.2. To guarantee that a lattice is distributive, it is both necessary and sufficient to ensure that it contains neither a pentagon nor a *diamond* [Gra78, Ch. II,

[2]Remarkably, a necessary and sufficient condition that that a lattice be distributive is that join correspond to union and meet to intersection. In other words, the only distributive lattices are those which, up to isomorphism, are lattices of subsets of a universal set, with union and intersection as join and meet, respectively. This is the classical representation theorem of Birkhoff and Stone, and is discussed in more detail in 1.1.3.

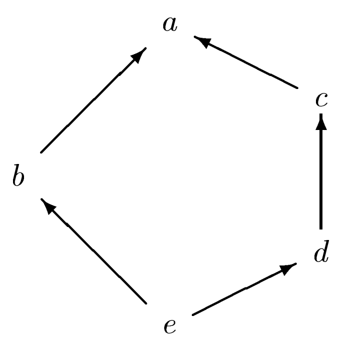

Figure 0.2: A generic pentagon in a lattice.

Sec. 1, Thm. 1]. A generic diamond is depicted in Figure 0.3.

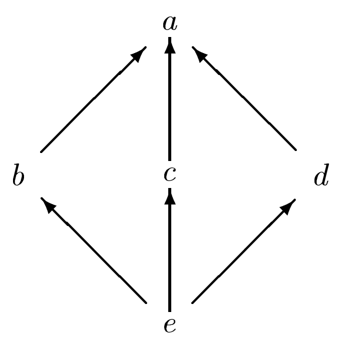

Figure 0.3: A generic diamond in a lattice.

The reason that nondistributive type hierarchies, such as the example just given, are not a problem in systems such as ALE, LKB, FTS, LOGIN, and LIFE, is that the full lattice operations are not interpreted in the fashion described. The initial hierarchy, such as that given in Figure 0.1, is interpreted not as the final type hierarchy, but rather as a *specification* for the construction of a more extensive type hierarchy. Indeed, all that is required in these systems is that the specification be a meet semilattice — that is, that any pair of elements have a greatest lower bound. Then, greatest lower bound is interpreted as intersection of object classes, but least upper bound is left uninterpreted. The "true" lattice — in which both meet and join have natural interpretations as intersection and union, respectively, of object classes — is constructed from the meets using a construction which builds the joins in a "free" fashion. From our initial type hierarchy, we build a new hierarchy consisting of all of the *order ideals* of the original hierarchy; that is, the set of all subsets which are closed under the "less than" relation. (In other words, S is an order ideal if $x \in S$ and $y \leq x$ implies $y \in S$.) It is well-known that the set of all order ideals of a finite poset forms a distributive lattice [DP90, 8.20]. Furthermore, this distributive lattice contains, as an embedded partially ordered set, the original poset **S**. Specifically, call an element x of a lattice *join-irreducible* if it is not the least element of the lattice (if indeed the lattice has a least element), and, whenever x may be written in the form $a \vee b$, then $x = a$ or $x = b$. The join-irreducible order ideals are in bijective correspondence with the elements of the poset itself [DP90, 8.20]. We illustrate by constructing the ideals of the pentagon of Figure 0.2. Figure 0.4 gives two "views" of this construction. On the left, we show the order ideals, with those which correspond to the original elements of the poset enclosed in a rectangle. On the right, we show the same lattice, but with a more informative labelling relative to the original poset, which shows only the maximal elements of the order ideals. Notice that neither $a = b \vee c$ nor $a = b \vee d$ holds in this new lattice. Rather, explicit new elements representing $b \vee c$ and $b \vee d$ have been constructed.

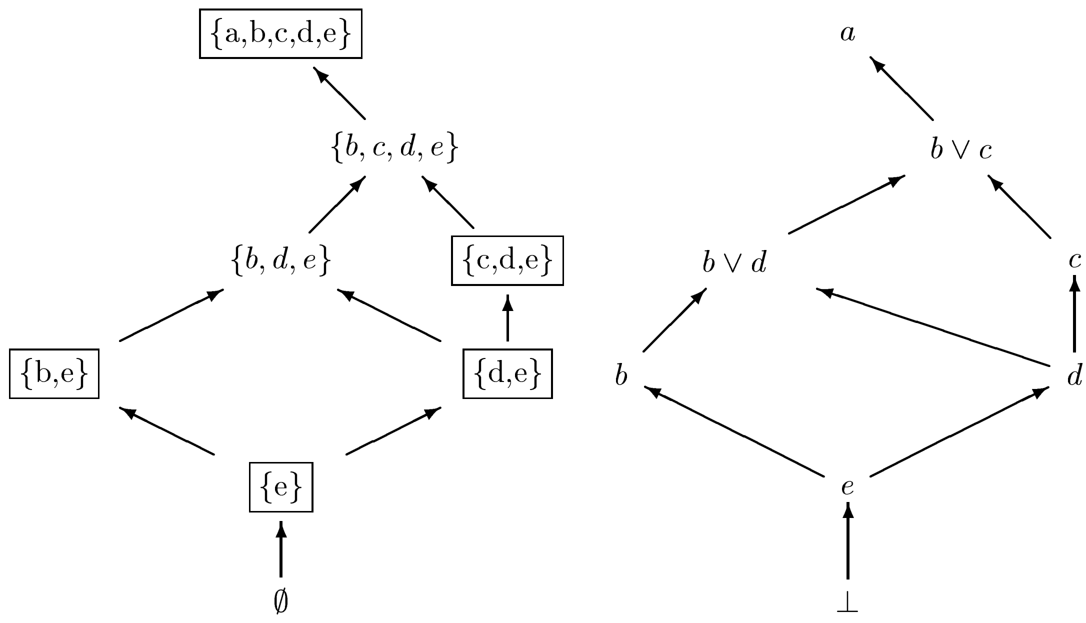

Figure 0.4: Two views of the "actual" distributive type hierarchy corresponding to the pentagon of Figure 0.2.

This construction has a further advantage. Even if the original type hierarchy did not have greatest lower bounds, it will create them automatically. This is illustrated in Figure 0.5. On the left is a simple specification without a greatest lower bound for b and c, and to its right is the corresponding distributive hierarchy.

0.3 The Power of CUF-like Specifications

Before going on, let us stop and take inventory of the features of the construction just described.

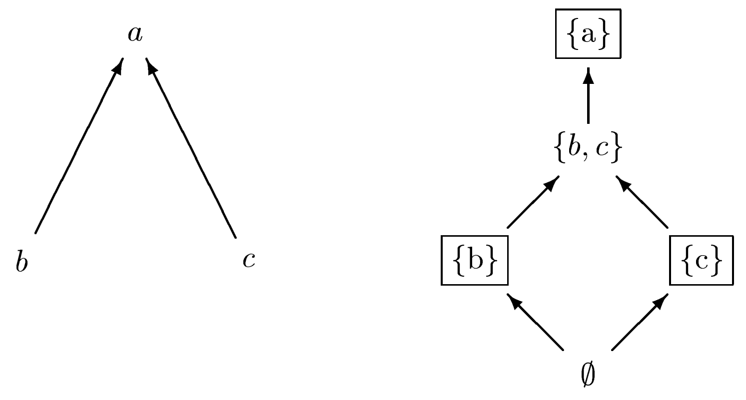

Figure 0.5: A type specification without a lower bound, and the order-ideal construction of the corresponding distributive hierarchy.

(a) Any distributive lattice may be represented in this fashion[3].

(b) Verifying that a specification is "legal" is trivial, since any partial order is legal.

(c) The representation of a distributive lattice in this fashion can be quite compact.

With respect to the last point, authors such as Carpenter [Car92b, pp. 15-17] argue that, for reasons of sparseness of the concept hierarchy, type hierarchies need not be distributive. However, what is (apparently) meant by such comments is that it is not practical to represent each and every type that would arise in an appropriate distributive hierarchy. Thus, the actual distributive hierarchy is encoded in the nondistributive fashion described above. Furthermore, meets represent actual logical meets. However, in such representations, the join operation does not necessarily correspond to logical disjunction of types. Rather, it simply corresponds to supremum amongst the explicitly specified types. Whether or not such a join operation could nonetheless be used in a productive way in such a system is an interesting question which, to our knowledge, has not been addressed. In all of the systems of which we are aware, it is not used explicitly at all.

Despite the attractive features of this method of specifying type hierarchies, there is a serious disadvantage. Principally, requiring the use of such specifications constrains the user completely in how a type hierarchy is specified. The construction is very rigid, and very asymmetric, in how it interprets a specification of type

[3]This follows from the duality between finite distributive lattices and finite partially ordered sets. See, for example, [DP90, 8.17 – 8.20]. Any finite partially ordered set may be turned into a lattice by adding a least element $\perp$ and a greatest element $\top$, in case these do not already exist. Then, define the greatest lower bound of two elements which previously had no lower bound to be $\perp$, and the least upper bound of two elements which previously had not upper bound to be $\top$. It is easy to show that the resulting structure is a lattice, and that it yields the same distributive lattice under this duality as did the originally partially ordered set.

inheritance. With this formalism, in some applications, it may be very difficult to correctly and efficiently express the type constraints which are desired. Hence, it is useful to have other means of specification at hand. Perhaps the most natural one is to forego any sort of clever encoding, and just specify the known constraints on the types, be they meet constraints, join constraints, or complement constraints. From there, it becomes the job of the system to check these constraints for consistency and to interpret them as a distributive lattice. This is precisely the tact taken with by the CUF system. Simply put, a CUF type hierarchy specification is just a set of constraints on how elements in the hierarchy should be related. CUF then attempts to build the most general hierarchy which satisfies those constraints, without making any particular assumptions. For this reason, we call the kind of type specifications which CUF admits *constraint-based specifications.*

Some examples will help illustrate the generality and advantages of this approach. Suppose that we are given three types, t_1, t_2, and t_3, and that they are related as shown in Figure 0.6. There are (at least) two possible interpretations to this

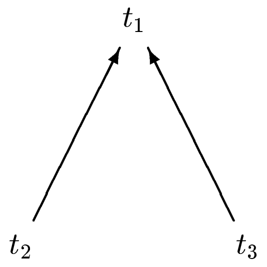

Figure 0.6: A simple upward type specification with two possible interpretations.

hierarchy. On the one hand, it may simply state that $t_3 < t_1$ and $t_2 < t_1$. This is the interpretation that the construction described in the previous subsection assigns to it. On the other hand, a specification with this graphical representation may provide more information; namely that $t_1 = t_2 \vee t_3$. In other words, it may actually say that $\mathcal{V}(t_1) = \mathcal{V}(t_2) \cup \mathcal{V}(t_3)$. A specification of the latter type is not at all unusual. It amounts to a disjunctive specification of a type. For example, in HPSG, an element of type *sign* must be either *lexical* or *phrasal*; there are no other possibilities. Thus, a program which is processing an object of type *sign* may decompose the processing into two cases, one for type *phrasal* and one for type *lexical.* Unfortunately, the method of defining a type hierarchy which was outlined in the previous section is incapable of recapturing such a specification. Thus, in systems using that approach, the semantics of this join must be realized outside of the type hierarchy. For example, in ALE, we may write a specification such as

```
bot sub [sign].
sign sub [phrasal, lexical].
```

but this only tells us that `phrasal` $<$ `sign` and `lexical` $<$ `sign` in the type hierarchy. The effect of the disjunctive processing must be recaptured by explicit statements in the ALE program itself; it cannot be an automatic consequence of the specification of the type hierarchy. On the other hand, in CUF, we may write a specification such as

```
sign = phrasal ; lexical.
```

The semicolon denotes disjunction, so this specification means precisely that the collection of objects of type `sign` is the union of those of type `phrasal` and those of type `lexical`. If we do not want this disjunctive specification, but only wish to specify that `phrasal` and `lexical` are subtypes of `sign`, then we may write

```
phrasal < sign.
lexical < sign.
```

Actually, in CUF, the most appropriate specification for this pair would be

```
sign = phrasal | lexical.
```

which means that `phrasal` and `lexical` are *disjoint* types which join to `sign`. That is, that both $\texttt{lexical} \vee \texttt{phrasal} = \texttt{sign}$ and $\texttt{lexical} \wedge \texttt{phrasal} = \bot$ hold.

Similar ideas hold for meets. Consider the description shown in Figure 0.7. Again, there are (at least) two possible interpretations. It may mean simply that

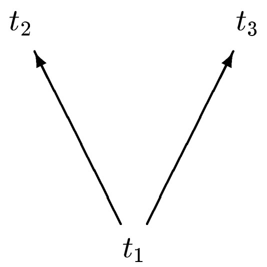

Figure 0.7: A simple downward type specification with two possible interpretations.

$t_1 < t_2$ and $t_1 < t_3$ in the type hierarchy, or it may mean that $t_1 = t_2 \wedge t_3$. In ALE, in the absence of further specifications about supertypes of t_1, the specification

```
bot sub [t2, t3].
t2  sub [t1].
t3  sub [t1].
```

can mean only that $\texttt{t1} = \texttt{t2} \wedge \texttt{t3}$. On the other hand, we may write the following specification in CUF.

```
t1 < t2.
t1 < t3.
```

It means only that `t1` is a subtype of both `t2` and `t3`. There is no implication that these are the only subtypes. To realize a similar situation in ALE, we would have to declare a greatest lower bound for `t2` and `t3`, and then make sure that `t1` is smaller than that bound, as illustrated in the following declaration.

```
bot sub [t2, t3].
t2 sub [t23].
t3 sub [t23].
t23  sub [t1].
```

This is illustrated graphically in Figure 0.8. On the other hand, CUF will auto-

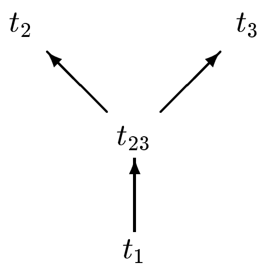

Figure 0.8: Adding a greatest lower bound.

matically generate the greatest lower bound, under the name `t2 & t3`, if needed. There is no need to identify it explicitly and give it a special name. Of course, the direct declaration of meets are also possible in CUF. To recapture the semantics of $\texttt{t1} = \texttt{t2} \wedge \texttt{t3}$, that is, that $\mathcal{V}(\texttt{t1}) = \mathcal{V}(\texttt{t2}) \cap \mathcal{V}(\texttt{t3})$ in the notation of the previous subsection, we may write the following CUF specification.

```
t1 = t2 & t3.
```

Sometimes, the relationship between types is not binary. For example, in at least some versions of HPSG, [PS87, p. 197], a *headed-structure* must be either a *head-complement-structure*, a *head-filler-structure*, or a *head-adjunct-structure*. This is suggested by the diagram of Figure 0.9. This relationship may be declared explicitly in CUF as

```
headed-structure = head-complement-structure ;
                   head-filler-structure;
                   head-adjunct-structure.
```

The semantics is $\mathcal{V}(\textit{headed-structure}) = \mathcal{V}(\textit{head-complement-structure}) \cup \mathcal{V}(\textit{head-filler-structure}) \cup \mathcal{V}(\textit{head-adjunct-structure})$. Again, the simple technique for describing distributive type hierarchies outlined in 0.2 cannot recapture this form of interaction. Therefore, in systems such as ALE, its semantics must be expressed with more direct representation outside of the type hierarchy description.

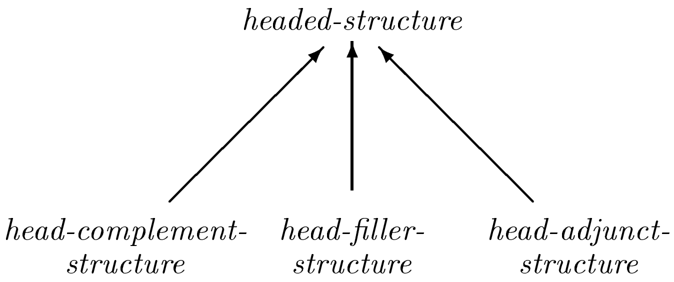

Figure 0.9: An upward type specification from HPSG.

Note also that there are no intermediate types for binary joins defined. They could be defined; in this case there would be three of them, *head-complement-structure* $\vee$ *head-filler-structure*, *head-complement-structure* $\vee$ *head-adjunct-structure*, and *head-filler-structure* $\vee$ *head-adjunct-structure*.

The problem with intermediate elements, and the contrast in expressive power between the type specification of CUF and the other systems, is most easily illustrated via an example of the meet of several elements. Figure 0.10 depicts a situation in which we want type t_1 to be the meet of the four types t_2, t_3, t_4, and t_5. In other

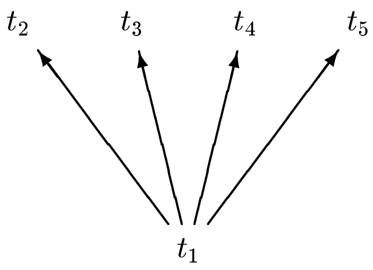

Figure 0.10: A downward type specification of four elements.

words, the semantics is $\mathcal{V}(t_1) = \mathcal{V}(t_2) \cap \mathcal{V}(t_3) \cap \mathcal{V}(t_4) \cap \mathcal{V}(t_5)$. In CUF, we may easily express this sort of relationship with the comparatively simple declaration

```
t1 = t2 & t3 & t4 & t5.
```

To realize the same sort of thing in ALE (or LIFE or LKB), we would have to introduce all of the (ten) intermediate types, and then express the relationship between them. In ALE, we would have to write something like that depicted in Figure 0.11. Clearly, this is much more tedious, and requires us to name and declare explicitly many new types that we may never use. The number of such types grows combinatorially; for a declaration expressing a type to be the intersection of n types, there will be a total of $\sum_{i=2}^{n-1} \binom{n}{i} = 2^n - (n+2)$ intermediate types. It is therefore an immense advantage not to have to specify them explicitly. In fact, for large expressions, it is a necessity. This is yet another advantage of the CUF formalism.

```
bot  sub [t2, t3, t4, t5].
t2   sub [t23, t24, t25].
t3   sub [t23, t34, t35].
t4   sub [t24, t34, t45].
t5   sub [t25, t35, t45].
t23  sub [t234, t235].
t24  sub [t234, t245].
t25  sub [t235, t245].
t35  sub [t235, t345].
t45  sub [t245, t345].
t234 sub [t1].
t235 sub [t1].
t245 sub [t1].
t345 sub [t1].
t1   sub [].
```

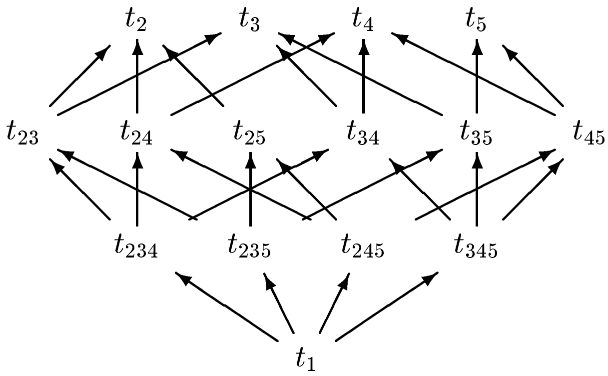

Figure 0.11: ALE declaration of the hierarchy of Figure 0.9 and the resulting completed type hierarchy.

The CUF type specification language allows one more form of construction that we have not mentioned. Namely, we may explicitly work with the complement of a type. If `t` is any type at all, then `~t` represents the *complement* of that type. That is to say, $\mathcal{V}(\texttt{t}) \cap \mathcal{V}(\texttt{\~{}t}) = \emptyset$ and $\mathcal{V}(\texttt{t}) \cup \mathcal{V}(\texttt{\~{}t}) = \mathcal{U}$. In ALE and the other languages mentioned, there is no facility at all for explicitly identifying complementary types.

0.4 Truly Nondistributive Type Hierarchies

A question which arises repeatedly when studying type hierarchies is whether or not they must be distributive. The answer to this question, of course, depends upon what such hierarchies are intended to represent. If each type represents a certain class of objects in some domain, and if meet corresponds to intersection of object classes and join corresponds to union of object classes, then the type hierarchy

must be distributive. This is a consequence of the classical Birkhoff representation theorem, which is discussed in 1.1.3 of this report. It is not clear, at least to this author, that there are conditions under which one would have a type hierarchy in which types would not be associated with object classes in this fashion. (Of course, a nondistributive lattice may be used to *represent* a distributive one, as outlined in 0.2, but that is a different issue.) Nonetheless, we do not claim to make any argument for or against modelling type classes distributively in this report. We simply present a theory which is valid under the assumption that a distributive hierarchy is desired.

0.5 Overview of this Report

The general philosophy of a constraint-based type specification language, such as that of CUF, is that only those relationships between the types which are required to hold need be specified. We start with a set of type names, and then we may express (almost) any Boolean constraint (based upon join, meet, and complement) between the types identified by these names. Indeed, complex expressions, such as

```
t = ((t1 & t2 & t3) ; (t4 & (~t5 ; t6)) & t7).
```

are allowed in CUF[4]. We have argued that this general type declaration system offers substantial advantages over the more restrictive methods employed by other systems. However, this increased power comes at a price. Namely, testing for consistency becomes a nontrivial task. In the simple technique described in Section 0.2, *any* poset whatever may be used to construct a distributive type hierarchy. There are no consistency questions of any form to worry about. In the more general approach of constraint-based specifications, however, we must address a consistency question. Namely, we need to determine whether or not the specification which is given may be extended to a distributive lattice. That this question is nontrivial is immediate. Indeed, we may specify any finite lattice in this language, and hence any finite nondistributive lattice. So there is definitely some testing which needs to be performed.

In the case that the partial specification is in fact total — that is, that it is specified as a lattice, then we can check in time $0(n^3)$, where n denotes the number of type names, whether or not it is distributive. Indeed, all we need do is test each ordered triple (a, b, c) of elements against the distributive law $(a \vee b) \vee (a \wedge c) = a \wedge (b \vee c)$ [Gra78, Ch. I, Sec. 4, Lem. 10]. However, if the specification is partial, then the testing process is much more complicated. A main result of this report is that this problem of testing for distributivity on partially specified structures is NP-complete, even under quite restrictive conditions about the constraints. This is true whether we talk of any extension at all (possibly non-injective, in which two names may be forced to represent the same set of objects), an injective extension, a free extension

[4]There are a few restrictions, imposed for practical reasons, but they are "inessential" from a mathematical point of view, and may be safely ignored from the point of view of a conceptual understanding. A more systematic overview of the type language for CUF may be found in the appendix of this report.

(one with the fewest constraints possible), or an extension to a Boolean lattice (*i.e.*, one with complements). Regarding the form of the constraints, even if the *fanout* — the number of disjuncts or conjuncts in any given declaration — is bounded by two, the problem is NP-complete, provided that the height of the specification — the length of the longest chain — is at least three. And if we allow a fanout of three, then even considering specifications of height two results in an NP-complete problem.

Beyond the complexity results, our main contribution is to provide a rigorous mathematical framework for studying constraint-based type specifications, and this allows us to demonstrate several important theoretical results. We show, for example, that in the absence of declarations of constant types (*i.e.*, types which cannot have nonempty subtypes), a consistent specification always has a *free extension* — that is, a natural one which imposes the minimum set of constraints possible. Thus, there never needs to be any "nondeterministic" choice about which types are collapsed and which are distinct; a type specification can always be expanded to a most general consistent distributive lattice.

Let us consider this freeness problem a bit more. The CUF system checks to see whether or not a type specification is consistent — that is, whether or not there is some assignment of nonempty sets of elements to each type (except for $\perp$) which makes the specification consistent. It is easy to see that this problem is equivalent to the satisfiability problem for certain classes of propositional Boolean formulas. CUF, however, does not check for so-called *separability* — that is, whether or not each type symbol may be assigned a subset of the universe which is distinct from the set assigned to each other type symbol. Thus, if we give CUF a specification which describes the diamond depicted in Figure 0.3, such as the specification shown below,

```
a = b ; c.
a = b ; d.
a = c ; d.
e = b & c.
e = b & d.
e = c & d.
```

it will accept this specification without complaint, even though all of the types must have the same denotation. We show that this separability problem is, up to deterministic polynomial-time equivalence, of the same complexity as the consistency problem. For the user, it would be very useful to know which types are collapsed and which are not, as two types which must always have the same denotation might well signal an error in the specification. However, this might not appear at first sight to be a well-defined question, because there might be some "nondeterminism" involved. That is, we might be able to separate `t1` and `t2`, but only at the expense of being unable to separate `t3` and `t4`, with the latter being separable if `t1` and `t2` are given the same denotation. Our theory shows that this cannot be the case — that is, that separability may be tested pairwise, and combined. Thus, if we have n type names, we may get away with $0(n^2)$ tests rather than 2^n to determine the complete separability picture. This "pairwise" separability idea also extends to types built up as expressions, such as `t1 & ~t2`, and it is this observation that allows us to build the free distributive lattice over a specification.

Unfortunately, if we allow constant types, the situation becomes much more complicated. In this case, we show that we may be forced to make nondeterministic choices in which elements are coalesced.

0.6 A Roadmap of this Report

Since this is quite a long report, we will give a brief overview of how it is organized.

Section 1 is devoted to the study of a formal theory of partially specified distributive structures. The intent is to be reasonably general, so, for example, no assumptions of finiteness are made, even though the type hierarchies which arise in actual applications generally are finite.

1.1 is a review of partially ordered sets and lattices, including distributive structures and structures with complements. The purpose of this subsection is largely to establish terminology, notation, and to point out the key results in the field which we use.

In 1.2, we turn to the issue of generalizing such structures in such a way that the operations (join and meet) are only partially defined. The work in this section may be viewed as a natural generalization of the idea of a *weak partial lattice*, as described in [Gra78, Ch. I, Sec. 5]. Our concepts are more general than those given in this reference in that we do not restrict attention to binary operations. Of course, when one is dealing with total associative operators, it suffices to consider binary operations, and this is precisely what is done in lattice theory. But, one may have partial structures in which the join of, say, three given elements is defined, without the joins of two-element subsets of this set being defined. Indeed, this kind of partial definition of varying arity occurs frequently in computational-linguistics applications, so this generalization is critical. We provide two extended concepts. With *bounded posets with partial operators (BPPO's)*, we work with a structure which has both an order structure and partial join and meet operations (with non-fixed arity) defined. With *generalized bounded weak partial lattices (GBWPL's)*, we work with structures in which everything is defined in terms of the partial join and meet operations. The latter concept is a direct generalization of weak partial latices. We show that the concepts of BPPO and of GBWPL are equivalent.

In 1.3, we define the notion of extension. Roughly, an extension of a partial structure $\mathbf{P}$ is a lattice $\mathbf{L}$ and a morphism $\eta : \mathbf{P} \to \mathbf{L}$. The type of extension is catalogued by two things: properties of the lattice $\mathbf{L}$ (such as distributivity and being Boolean), and by properties of the morphism η (such as being injective, being an embedding, or having certain universal properties). A key point is that there is no single notion of extension which works for all applications. Each type of extension has advantages and disadvantages. A main result of this section is that any BPPO has an injective extension (in which η is an injective function.) We emphasize that $\mathbf{L}$ is not required to be distributive in this result.

In 1.4, we turn to the problem of extensions in which the lattice $\mathbf{L}$ is required to be distributive. In contradistinction to the result of 1.3, it is not the case that an arbitrary BPPO has a distributive extension. Also, it may have a distributive extension without having an injective one, and universal extensions need not be injective. The main technique of characterization is that of *ideal binary decompositions*. Roughly, what the theory provides is that, to completely characterize the distributive extensions of a BPPO $\mathbf{P}$, it suffices to characterize the morphisms $\mathbf{P} \to \mathbf{2}$, with $\mathbf{2}$ representing the (unique up to isomorphism) two-element lattice. A further result, as mentioned previously, is that, if there is any nontrivial extension at all, then there is a "maximal" extension. No nondeterministic choice is involved.

In 1.5, the complications which arise when one introduces atoms (constant types) into the type hierarchy are investigated. Particularly, we show that there no longer need be a maximal extension.

Section 2 turns to the issue of computational complexity. We know that not every BPPO has a distributive extension, so we naturally ask how difficult it is to decide whether or not is has one. Of course, in this section, we restrict our attention to finite structures. We break the investigation into two steps. The first involves determining whether or not a prespecification defines a BPPO, the second involves determining whether or not a BPPO has an extension of a given kind.

In 2.1, we develop the idea of a *prespecification*. When a type hierarchy is specified in practice, the complete algebraic structure of a BPPO is not given by the user. Rather, a skeletal specification, which may be extended to a BPPO, is given. The extension involves applying associative and other laws which hold in any BPPO. In terms of the general type of statements involved, our notion of a prespecification corresponds quite closely to that of a CUF program. In 2.1, we show that if a prespecification defines a BPPO, then there is a natural least (in terms of the size of the extension) BPPO which it defines. We also show that it may be decided in deterministic polynomial time whether or not such an extension of a prespecification to a BPPO exists.

In 2.2, we investigate the complexity of determining whether or not a BPPO has a distributive extension. The crux of the results is that the question is NP-complete, regardless of what kind of extension (injective, universal, arbitrary) we are seeking, so long is it is required to be distributive. For the case of an arbitrary extension, this question may be easily viewed as a satisfiability problem for propositional formulas. However, to determine whether or not there is an injective extension, some rather nontrivial transformations from so-called separability problems are involved, and the main mathematical contribution is to show how these transformations are realized.

In 2.3, we refine the problems under consideration some, by looking at the complexity of special cases in which the size of various parameters characterizing the prespecification (join fanout, meet fanout, length of the longest path in the hierarchy) are restricted. We show that as long as all of these parameters are at least two, and at least one of them is at least three, then the problem remains NP-complete. This effectively demonstrates that placing reasonable restrictions on these parameters is not likely to lead to tractable cases.

In 2.4, we do identify some cases which are tractable. In one case, we show that if meet fanout, join fanout, and height of the hierarchy are all restricted to be no more than two, then the problem may be solved in deterministic polynomial time. However, this case is admittedly not of much practical interest. A second case is possibly of more interest. If the specifications involve only meets or only joins (but not both), then the decision problem is also solvable in deterministic polynomial time. However, in this case, we are effectively reduced to the type of specification outlined in 0.2.

Finally, in 2.5, we examine how the complexity is affected by the introduction of atoms and by requiring intermediate levels of injectivity. Nothing becomes worse than NP-complete, although some of the problems identified in 2.4 may not remain solvable in deterministic polynomial time.

In Section 3, we present some conclusions and suggestions for further directions.

In an appendix, we present a more complete description of the CUF type specification system, and show how it is formally related to our work. Finally, an index is provided to make it easier for the reader to find definitions of key concepts and the meaning of notational symbols.

0.7 Prerequisites

We expect the reader to be familiar with the general ideas of lattice theory, as may be found in [Gra78]. However, we do not use any serious theorems or definitions without at least providing a complete reference. Also, we use very elementary language of category theory, because it allows us to unify otherwise cumbersome notions. The definitions of category, morphism and isomorphism are all that we use, and even the most elementary reference, such as [Wal91], will prove far more than adequate. However, we use [HS73] as our standard for notation and terminology.

For the complexity theory part, we assume familiarity with the basic notation, terminology and results in the theory of algorithm analysis, as may be found in [CLR90], and NP-completeness, as may be found in [GJ79] or [AHU74].

1. The Theory of Distributive Partial Structures

In this section, we develop the general theory of distributive partial structures — that is, a theory of partially specified lattice-like structures, and the conditions under which they may be extended to bounded distributive lattices and bound Boolean lattices of various kinds. The results do not really depend upon finiteness in any way, and restricting our attention to finite structures would only simplify a few definitions and proofs slightly. Therefore, we present the results in full generality, without making unnecessary assumptions.

1.1 Review of Total Order-Theoretic Structures

We begin with a review of total order-theoretic structures, including bounded posets and various forms of lattices. Although we expect that the reader is familiar with these ideas, it is important to present them explicitly in order that we establish both a consistent notation and a visible context for the various generalizations to partial structures that we develop. We start with bounded posets.

1.1.1 Bounded Posets and Notational Conventions A *bounded poset* is a four-tuple $\mathbf{P} = (P, \leq, \top, \bot)$ such that

(bp-i) P is a nonempty set (the *underlying set*).

(bp-ii) $\leq$ is a partial order (reflexive, transitive, and antisymmetric) on P.

(bp-iii) $\bot \in P$ is called the *least element* and satisfies $\bot \leq p$ for all $p \in P$.

(bp-iv) $\top \in P$ is called the *greatest element* and satisfies $p \leq \top$ for all $p \in P$.

(bp-v) $\bot$ and $\top$ are distinct elements.

Even when working with several bounded posets, we will usually use the same symbols for the order relation ($\leq$), the least element ($\bot$), and the greatest element ($\top$) in each. When it is absolutely necessary to use distinct symbols, we will use subscripts or other indicators. Also, to avoid the need to write out detailed specifications at every turn, we shall adopt the convention that when such a poset is named by a boldface letter (*e.g.*, $\mathbf{P}$), then the underlying set is named by the same letter in roman italic font (*e.g.*, P). We adopt similar conventions for the other types of structures (lattices and the like) which we define and use later on in this report. Also, as is customary, we write $x < y$ to mean $x \leq y$ and $x \neq y$.

In a bounded poset $\mathbf{P} = (P, \leq, \top, \bot)$, an *upper bound* of a set $S \subseteq P$ is any $p \in P$ such that $s \leq p$ for all $s \in S$. Note that $\top$ is an upper bound for any $S \subseteq P$. The *least upper bound* (*lub*) of S, when it exists, is the unique upper bound $p \in P$ such that $p \leq r$ for all r which are upper bounds of S. Dually, a *lower bound* of S is any $p \in P$ such that $p \leq s$ for all $s \in S$, and the *greatest lower bound* (*glb*) of S, when it exists, is the unique lower bound $p \in P$ such that $r \leq p$ for all r which are lower bounds of S. Note that $\bot$ is a lower bound for any $S \subseteq P$. For a given set S, we write lub(S) (resp. glb(S)) to denote the least upper bound (resp. greatest lower bound) of S, when it exists.

Let $\mathbf{P}_1 = (P_1, \leq, \top, \bot)$ and $\mathbf{P}_2 = (P_2, \leq, \top, \bot)$ be bounded posets. A *morphism* $f : \mathbf{P}_1 \to \mathbf{P}_2$ is a function $f : P_1 \to P_2$ such that the following conditions are satisfied.

(bpmor-i) $f(\bot) = \bot$ and $f(\top) = \top$.

(bpmor-ii) For all $p, q \in P_1$ with $p \leq q$, $f(p) \leq f(q)$.

We let **BdPos** denote the category of bounded posets, with morphisms as defined above. When we speak of *isomorphisms* of bounded posets, or of algebraic structures more generally, we shall always use the categorical definition [HS73, 5.13]. That is, a morphism $f : \mathbf{P}_1 \to \mathbf{P}_2$ is an isomorphism if there is a morphism $g : \mathbf{P}_2 \to \mathbf{P}_1$ such that $g \circ f$ is the identity on $\mathbf{P}_1$ and $f \circ g$ is the identity on $\mathbf{P_2}$. In the case of bounded posets, it is easy to see that the morphism $f : \mathbf{P}_1 \to \mathbf{P}_2$ is an isomorphism iff the underlying $f : P_1 \to P_2$ is bijective and satisfies $f(x) \leq f(y) \Rightarrow x \leq y$.

1.1.2 Bounded Lattices and Related Concepts The simplest definition of a bounded lattice is that it is a bounded poset $\mathbf{P} = (P, \leq, \top, \bot)$ for which any two elements have a least upper bound and a greatest lower bound. The join operation $\vee : P \times P \to P$ and the meet operation $\wedge : P \times P \to P$ are then defined by $p \vee q = \mathrm{lub}(\{\mathrm{p}, \mathrm{q}\})$ and $p \wedge q = \mathrm{glb}(\{\mathrm{p}, \mathrm{q}\})$ [Gra78, p. 3]. Unfortunately, this definition will not be compatible with our definitions of partial operations (unless we were to work with two distinct partial orders). Therefore, we prefer the definition of a bounded lattice as an algebra. More precisely, we define a *bounded lattice* to be a five-tuple $\mathbf{L} = (L, \top, \bot, \vee, \wedge)$ in which:

(blat-i) L is a nonempty set (the *underlying set*).

(blat-ii) $\vee : L \times L \to L$ and $\wedge : L \times L \to L$ are idempotent, commutative, and associative operations.

(blat-iii) For all $x, y \in L$, $x \wedge (x \vee y) = x$ and $x \vee (x \wedge y) = x$. (These are the so-called *absorption identities*.)

(blat-iv) $\bot, \top \in L$ are such that for all $x \in L$, $x \vee \bot = x$ and $x \wedge \top = x$.

It is well known that this definition is equivalent to the characterization in terms of bounded posets, and that we may recover the partial order via $x \leq y$ iff $x \vee y = y$ [Gra78, Ch. I, Sec. 1, Thm. 1].

Let $\mathbf{L}_1 = (L_1, \top, \bot, \vee, \wedge)$ and $\mathbf{L}_2 = (L_2, \top, \bot, \vee, \wedge)$ be bounded lattices. A *morphism* $f : \mathbf{L}_1 \to \mathbf{L}_2$ is a function $f : L_1 \to L_2$ such that

(blmor-i) $f(\bot) = \bot$ and $f(\top) = \top$.

(blmor-ii) For all $x, y \in L_1$, $f(x \vee y) = f(x) \vee f(y)$ and $f(x \wedge y) = f(x) \wedge f(y)$.

If $f : \mathbf{L}_1 \to \mathbf{L}_2$ is an injective morphism, then it is automatically an *embedding*. That is to say, the image $f(L_1)$, under the operations inherited from $\mathbf{L}_2$, is isomorphic to the lattice $\mathbf{L_1}$ itself. This is a consequence of a more general theorem from universal algebra [Gra68, Ch. I, Sec. 7, Lem. 3]. In particular, bijective morphisms are always isomorphisms. We remark, however, that this result does not extend to bounded posets, nor to the various forms of partial structures that we discuss in the next section.

Let $\mathbf{L} = (L_1, \top_1, \bot_1, \vee_1, \wedge_1)$ and $\mathbf{L} = (L_2, \top_2, \bot_2, \vee_2, \wedge_2)$ be bounded lattices. $\mathbf{L}_1$ is a *bounded sublattice* of $\mathbf{L}_2$ if $L_1 \subseteq L_2$, $\bot_1 = \bot_2$, $\top_1 = \top_2$, and for every $x, y \in L_1$, $x \vee_1 y = x \vee_2 y$ and $x \wedge_1 y = x \wedge_2 y$. Note particularly that the greatest and least elements must be preserved.

The bounded lattice $\mathbf{L} = (L, \top, \bot, \vee, \wedge)$ is *distributive* if it satisfies any one of the following equivalent conditions [Gra78, Ch. I, Sec. 4, Lem. 10].

(dist-i) For all $x, y, z \in L$, $(x \wedge y) \vee (x \wedge z) = x \wedge (y \vee z)$.

(dist-ii) For all $x, y, z \in L$, $(x \vee y) \wedge (x \vee z) = x \vee (y \wedge z)$.

(dist-iii) For all $x, y, z \in L$, $((x \vee y) \wedge z) \vee (x \vee (y \wedge z)) = x \vee (y \wedge z)$.

A *complement* of an element $x \in L$ is a $y \in L$ such that $x \vee y = \top$ and $x \wedge y = \bot$. $\mathbf{L}$ is *complemented* if every $x \in L$ has a complement. A *Boolean lattice* is a distributive, complemented lattice. It is well-known that, in a distributive lattice, an element can have at most one complement [Gra78, Ch. I, Sec. 6, Lem. 1]. A *Boolean algebra* is a Boolean lattice augmented by a specific operation for the complement. More precisely, a *Boolean algebra* is a six-tuple $\mathbf{L} = (L, \top, \bot, \vee, \wedge, \prime)$ in which

(ba-i) $(L, \vee, \wedge, \bot, \top)$ is a Boolean lattice.

(ba-ii) $\prime : L \to L$ is the complement operation.

It is easy to verify that if $f : \mathbf{L}_1 \to \mathbf{L}_2$ is a surjective morphism of lattices and $\mathbf{L}_1$ is distributive (resp. Boolean), then so too is $\mathbf{L}_2$. Furthermore, complements are preserved under morphisms, so that a morphism of Boolean lattices is also a morphism of Boolean algebras.

We let **BdLat**, **BdDiLat**, and **BoolLat** denote the categories of bounded lattices, bounded distributive lattices, and bounded Boolean lattices, respectively.

The *dual* of the bounded lattice $\mathbf{L} = (L, \top, \bot, \vee, \wedge)$ is the structure $\mathsf{Dual}(\mathbf{L}) = (L, \bot, \top, \wedge, \vee)$. In other words, join becomes, meet, meet becomes join, and top and bottom are switched. The dual of a lattice is itself a lattice, and is distributive (resp. Boolean) iff the original lattice is. Note, however, that we usually do **not** use these reversed symbol conventions when working with the dual of a lattice. More specifically, we will use $\bot$, $\top$, $\vee$, $\wedge$ to denote the least element, greatest element, join operation, and meet operation, respectively, in the dual. When dealing with duals, we will always make it clear from the context exactly of which operations we speak.

There is a special Boolean lattice which we will use frequently in this work. We let $\mathbf{2}$ denote the Boolean lattice whose underlying set is just the two elements $\{\bot, \top\}$.

We will also have occasion to work with products of bounded lattices in this work. Given an indexed family $\{\mathbf{L}_i \mid i \in I\}$ of bounded lattices, their product is the lattice $\prod_{i\in I} \mathbf{L}_i = (\prod_{i\in I} L_i, \top, \bot, \vee, \wedge)$. Elements are just I-indexed tuples, with the i^{th} entry coming from L_i. We often write $\langle x_i \rangle_{i\in I}$ for such an I-tuple. The least element is the tuple with every position $\bot$, and the greatest element is the tuple with every position $\top$. Join and meet are defined componentwise. Given a subset $J \subseteq I$, we define the projection $\pi_J : \prod_{i\in I} \mathbf{L}_i \to \prod_{j\in J} \mathbf{L}_j$ to be the function which preserves the entries indexed by J and discards the others. If J is a singleton $\{j\}$, we may write π_j for $\pi_{\{j\}}$. It is easy to see that the product of distributive (resp. Boolean) lattices is distributive (resp. Boolean), as is any projection. Again, these results are consequences of more fundamental results from universal algebra. See [Gra68] for details.

A Boolean lattice is called *perfect* if it is isomorphic to a product of copies of $\mathbf{2}$. Instead of writing $\Pi_{i\in I}\mathbf{2}$, we may write $\mathbf{2}^I$. It is not the case that every Boolean lattice is perfect; rather, the perfect ones are those that have the additional properties of being *complete* and *completely distributive* [Bir67, Ch. 5, Thm. 17]. However, it is the case that every *finite* Boolean algebra is perfect [Gra78, Ch. 2, Sec. 1, Cor. 12]. We let **PerfBoolLat** denote the category of all perfect Boolean lattices.

1.1.3 Bit-Vector Lattices Let S be a set. A function $S \to \{0,1\}$ is called a *bit vector* over S, and we write $\mathsf{BitVec}(S)$ for the set of all such bit vectors. We define the *full bit-vector lattice* over S to be $\mathbf{BitVec}(S) = (\mathsf{BitVec}(S), \mathbf{1}, \mathbf{0}, \vee, \wedge)$, with $\mathbf{0}$ the function which maps $s \mapsto 0$ for each $s \in S$, and $\mathbf{1}$ the function which maps $s \mapsto 1$ for each $s \in S$. The operation $\vee : \mathsf{BitVec}(S) \times \mathsf{BitVec}(S) \to \mathsf{BitVec}(S)$ is bitwise logical disjunction; For $f, g \in \mathsf{BitVec}(S)$, $s \in S$, $(f \vee g)(s) = \max(\{f(s), g(s)\})$. Similarly, $\wedge : \mathsf{BitVec}(S) \times \mathsf{BitVec}(S) \to \mathsf{BitVec}(S)$ is bitwise logical conjunction; $(f \wedge g)(s) = \min(\{f(s), g(s)\})$. It is easy to see that, for any set S, $\mathbf{BitVec}(S)$ is naturally isomorphic to the perfect Boolean lattice $\mathbf{2}^S$, just by replacing 0 by $\perp$ and 1 by $\top$ in each S-tuple.

A *bounded bit-vector lattice* over S is any bounded sublattice of $\mathbf{BitVec}(S)$ which contains both $\mathbf{0}$ and $\mathbf{1}$; in particular these elements must be the least and greatest elements, respectively, of the sublattice. The set S is called the *index set* of the lattice.

It is easy to view a bounded bit-vector lattice as a lattice of subsets of S. Indeed, identify the bit vector f with $\{s \in S \mid f(s) = 1\}$. Then $\vee$ translates to set-theoretic union, $\wedge$ to set-theoretic intersection, $\mathbf{0}$ to the empty set, and $\mathbf{1}$ to S. What we obtain is the lattice associated with a *ring of sets*, that is a collection of subsets of a set S which is closed under union and intersection. Our boundedness condition yields a *bounded* ring of sets; that is, a ring which contains both $\emptyset$ and the full set S. A classical result of G. Birkhoff states that any distributive lattice may be represented by a ring of sets [Gra78, Ch. 2, Sec. 1, Thm. 19]. Similarly, according to a result of M. H. Stone, any Boolean lattice may be represented by a *field of sets*, which is a ring of sets which is also closed under relative complementation with the base set S [Gra78, Ch. 2, Sec. 1, Cor. 21]. Adding bounds is a trivial modification to these results. We summarize the key results in the following.

1.1.4 Proposition — representation of distributive and Boolean lattices *Let $\mathbf{L}$ be a bounded lattice.*

(a) *$\mathbf{L}$ is distributive iff it is isomorphic to a bounded bit-vector lattice.*

(b) *If $\mathbf{L}$ is finite, then it is Boolean iff it is isomorphic to $\mathbf{BitVec}(S)$ for some finite set S.*

(c) *If $\mathbf{L}$ is distributive, then it may be embedded into a perfect Boolean lattice.* □

1.1.5 Distributive extension categories This report is concerned with extending partial structures to total distributive ones. The categories of distributive lattices which we shall consider extending to are **BdDiLat**, **BoolLat**, and **PerfBoolLat**. Collectively, we shall call these categories the *distributive extension categories.*

1.2 Structures with Partial Operations

We now turn to the axiomatization of lattice-like structures with partial operations. Since we are interested in modelling n-ary specifications, for arbitrary finite n, such as CUF specifications like the example below,

```
t = t1 & t2 & .. & tn.
```

these partial operations are **not** constrained to be binary. Rather, we allow general partial join and meet operations, which range over finite subsets of the underlying set.

We provide two equivalent characterizations. In the first, we start with a bounded poset and add the partial generalized join and meet operations on top of it. In the second, we eschew an explicit presentation of the underlying partial order entirely, and express everything in terms of partial operations and axioms. This latter characterization, when restricted to binary partial operations, yields precisely the (bounded) weak partial lattices of Grätzer [Gra78, p. 41].

1.2.1 Notation There is some notation which we use throughout, and we collect it here. Whenever we write $A \subseteq_f B$, we mean that A is a *finite* subset of B. And, we let $\mathcal{P}_f(P)$ denote the set of all *finite* subsets of P, while $\mathcal{P}(P)$ denotes the full powerset of P, consisting of all subsets, finite or not.

Given a partial function $f : A \to B$ we write $f(a)\downarrow$ to mean that f is defined on the argument a. When we speak of a partial function, we do not discount the possibility that the function is in fact defined on all of its arguments. In other words, total functions are partial in our terminology.

1.2.2 Bounded Posets with Partial Operations A *bounded poset with partial operations* (BPPO) is a 6-tuple $\mathbf{P} = (P, \leq, \top, \bot, \bigvee, \bigwedge)$ in which

(bppo-i) $(P, \leq, \top, \bot)$ is a bounded poset, called the *underlying poset.*

(bppo-ii) $\bigvee : \mathcal{P}_f(P) \to P$ is partial a operation, called the *generalized join.*

(bppo-iii) $\bigwedge : \mathcal{P}_f(P) \to P$ is partial a operation, called the *generalized meet.*

The operations $\bigvee$ and $\bigwedge$ are subject to the following conditions.

(bppo-iv) For any $S \subseteq_f P$, if $\bigvee S\downarrow$, then $\bigvee S = \text{lub(S)}$. Dually, if $\bigwedge S\downarrow$, then $\bigwedge S = \text{glb(S)}$.

(bppo-v) For $S \subseteq_f P$, if $\text{lub(S)}\downarrow$ and $\text{lub(S)} \in \text{S}$, then $\bigvee S\downarrow$. Dually, if $\text{glb(S)}\downarrow$ and $\text{glb(S)} \in \text{S}$, then $\text{glb(S)}\downarrow$.

(bppo-vi) $\bigvee \emptyset = \bot$ and $\bigwedge \emptyset = \top$.

(bppo-vii) If $S_1, S_2, \ldots, S_n \subseteq_f P$ are such that $\bigvee S_i\downarrow$ for all i, $1 \leq i \leq n$, then $\bigvee(\bigcup_{i=1}^n S_i)\downarrow$ iff $\bigvee\{\bigvee S_1, \bigvee S_2, \ldots \bigvee S_n\}\downarrow$, and $\bigvee\{\bigvee S_1, \bigvee S_2, \ldots \bigvee S_n\} = \bigvee(\bigcup_{i=1}^n S_i)$ in this case. Dually, if $\bigwedge S_i\downarrow$ for all i, $1 \leq i \leq n$, then $\bigwedge(\bigcup_{i=1}^n S_i)\downarrow$ iff $\bigwedge\{\bigwedge S_1, \bigwedge S_2, \ldots \bigwedge S_n\}\downarrow$, and $\bigwedge\{\bigwedge S_1, \bigwedge S_2, \ldots \bigwedge S_n\} = \bigwedge(\bigcup_{i=1}^n S_i)$ in this case. These are called the *generalized associativity laws.* .

Condition (bppo-iv) requires that, if a generalized join (resp. meet) exists, then it must be the lub (resp. glb) in the partial order. Note, however, that we do *not* demand that subsets which have an lub (resp. glb) in the partial order have a join (resp. meet). We simply require that, whenever they are defined, joins and meets agree with these bounds. Condition (bppo-v) says that any finite subset of P which contains its lub (resp. glb) has this lub (resp. glb) as its join (resp. meet). Note well that the lub (resp. glb) must already be in S; it is not enough that it exist in P. Condition (bppo-vi) is a natural rule for the empty set. Condition (bppo-vii), the natural associativity laws, just say that the join and meet are associative in a more general context. For example, given the following CUF specifications,

```
t1 = t11 & t12 & t13.
t2 = t21 & t22 & t23.
t3 = t31 & t32 & t33.
t4 = t11 & t12 & t13 & t21 & t22 & t23 & t31 & t32 & t33.
```

we may write these in our notation as $\bigwedge(\{\mathtt{t11}, \mathtt{t12}, \mathtt{t13}\}) = \mathtt{t1}$, $\bigwedge(\{\mathtt{t21}, \mathtt{t22}, \mathtt{t23}\}) = \mathtt{t2}$, $\bigwedge(\{\mathtt{t31}, \mathtt{t32}, \mathtt{t33}\}) = \mathtt{t3}$, and $\bigwedge(\{\mathtt{t11}, \mathtt{t12}, \mathtt{t13}, \mathtt{t21}, \mathtt{t22}, \mathtt{t23}, \mathtt{t31}, \mathtt{t32}, \mathtt{t33}\}) = \mathtt{t4}$. The generalized associativity law for meets then tells us that we must have $\bigwedge(\{\mathtt{t1}, \mathtt{t2}, \mathtt{t3}\}) = \mathtt{t4}$,

Let $\mathbf{P}_1 = (P_1, \leq, \top, \bot, \bigvee, \bigwedge)$ and $\mathbf{P}_2 = (P_2, \leq, \top, \bot, \bigvee, \bigwedge)$ be BPPO's. A *morphism* $f : \mathbf{P_1} \to \mathbf{P_2}$ is a poset morphism $f : (P_1, \leq, \top, \bot) \to (P_2, \leq, \top, \bot)$ subject to the following additional constraint.

(bppomor-i) For $S \subseteq_f P_1$, if $\bigvee S\!\downarrow$, then $\bigvee(f(S))\!\downarrow$ and $f(\bigvee S) = \bigvee(f(S))$. Dually, if $\bigwedge S\!\downarrow$, then $\bigwedge(f(S))\!\downarrow$ and $f(\bigwedge S) = \bigwedge(f(S))$[5].

Notice that a morphism automatically preserves $\bot$ and $\top$. For example, $f(\bot) = f(\bigvee \emptyset) = \bigvee f(\emptyset) = \bigvee \emptyset = \bot$.

We let **BPPO** denote the category of BPPO's.

We now turn to an alternate representation of a BPPO which eschews the explicit use of a partial order.

1.2.3 Generalized Bounded Weak Partial Lattices

In [Gra78, Ch. I, Sec. 5], Grätzer introduces the notion of a *weak partial lattice*. We provide a definition which generalizes his concept to meet and join operations with more than two arguments. A *generalized bounded weak partial lattice* (*GBWPL*) is a five-tuple $\mathbf{L} = (L, \top, \bot, \bigvee, \bigwedge)$ in which

(gbwpl-i) L is a set (the *underlying set*).

(gbwpl-ii) $\bigvee : \mathcal{P}_f(L) \to L$ is partial operation, called the *generalized join*.

(gbwpl-iii) $\bigwedge : \mathcal{P}_f(L) \to L$ is partial operation, called the *generalized meet*.

(gbwpl-iii) $\bot, \top \in L$ with $\bot \neq \top$.

[5] $f(S)$ denotes $\{f(s) \mid s \in S\}$.

The operations $\bigvee$ and $\bigwedge$ are subject to the following conditions.

(gbwpl-iv) $\bigvee\emptyset = \bot$, $\bigwedge\emptyset = \top$. For all $a \in L$, $\bigvee\{a\} = a$, $\bigwedge\{a\} = a$, $\bigvee\{a, \bot\} = a$, $\bigwedge\{a, \top\} = a$,

(gbwpl-v) The generalized associativity laws stated in (bppo-vii) (with P replaced by L.)

(gbwpl-vi) If $S \subseteq_f L$ with $\bigvee S\downarrow$, then for all $a \in S$, $\bigwedge\{a, \bigvee S\}\downarrow$ with $\bigwedge\{a, \bigvee S\} = a$. Dually, for all $a \in S$, $\bigvee\{a, \bigwedge S\}\downarrow$ with $\bigvee\{a, \bigwedge S\} = a$. These are called the *generalized absorption identities.*

In (gbwpl-iv), the condition $\bigvee\{a\} = a$ is just a restatement of the idempotency of the join, which becomes $a \vee a = a$ in a lattice. The condition $\bigvee\{a, \bot\} = a$ just states that $\bot$ is the least element. The other two new conditions are dual. In (gbwpl-vi), The condition $\bigwedge\{a, \bigvee S\} = a$ for $a \in S$ and $\bigvee S\downarrow$ is a generalization of the absorption identity $a \wedge (a \vee b) = a$ of an ordinary lattice [Gra78, Cond. (L4), p. 4]. The other condition is dual.

Let $\mathbf{L}_1 = (L_1, \top, \bot, \bigvee, \bigwedge)$ and $\mathbf{L}_2 = (L_2, \top, \bot, \bigvee, \bigwedge)$ be GBWPL's. A *morphism* $f : \mathbf{L_1} \to \mathbf{L_2}$ is a function $f : L_1 \to L_2$ subject to the following additional constraints.

(gbwplmor-i) For $S \subseteq_f L_1$, if $\bigvee S\downarrow$, then $\bigvee(f(S))\downarrow$ and $f(\bigvee S) = \bigvee(f(S))$. Dually, if $\bigwedge S\downarrow$, then $\bigwedge(f(S))\downarrow$ and $f(\bigwedge S) = \bigwedge(f(S))$.

We let **GBWPL** denote the category of GBWPL's.

1.2.4 Relationship between BPPO's and GBWPL's The concepts of BPPO and GBWPL are equivalent, as we shall prove in the next proposition. First of all, we need to define the appropriate transformations. To obtain a GBWPL from a BPPO, we just "forget" the order structure. Given a BPPO $\mathbf{B} = (B, \leq, \top, \bot, \bigvee, \bigwedge)$, the associated GBWPL is just $\mathbf{L} = (L, \top, \bot, \bigvee, \bigwedge)$, which we denote by $\mathsf{GBWPL}(\mathbf{B})$. The underlying function of a **BPPO** morphism $f : \mathbf{B}_1 \to \mathbf{B}_2$ is the same as the underlying function of the corresponding **GBWPL** morphism $\mathsf{GBWPL}(f) : \mathsf{GBWPL}(\mathbf{B}_1) \to \mathsf{GBWPL}(\mathbf{B}_2)$. Thus, as a simplification of notation, we use the same symbol f for the morphism in each case. In the other direction, things are a bit more complicated. We start with a GBWPL $\mathbf{L} = (L, \top, \bot, \bigvee, \bigwedge)$. Define the relation $\leq$ on L by $a \leq b$ iff either there is a subset $S \subseteq_f L$ with $\bigvee S\downarrow$, $a \in S$, and $b = \bigvee S$, or else such an S with $\bigwedge S\downarrow$, $b \in S$, and $a = \bigwedge S$. The associated BPPO is then just $\mathbf{L} = (L, \leq, \top, \bot, \bigvee, \bigwedge)$, and is denoted $\mathsf{BPPO}(\mathbf{L})$. Again, the underlying function of a **GBWPL** morphism f is the same as the underlying function of the associated **BPPO** morphism $\mathsf{BPPO}(f)$, and so we use the same symbol f in each case. We will show that $\mathsf{BPPO} : \mathbf{GBWPL} \to \mathbf{BPPO}$ and $\mathsf{GBWPL} : \mathbf{BPPO} \to \mathbf{GBWPL}$ are functors which are mutual inverses, in the sense that $\mathsf{GBWPL} \circ \mathsf{BPPO}$ is the identity on **GBWPL** and $\mathsf{BPPO} \circ \mathsf{GBWPL}$ is the identity on **BPPO**.

1.2.5 Proposition *Let* $\mathbf{L} = (L, \top, \bot, \bigvee, \bigwedge)$ *be a GBWPL. Then* $\mathsf{BPPO}(\mathbf{L})$ *is a BPPO. Furthermore, if* $f : L_1 \to L_2$ *is the underlying function of a morphism between two GBWPL's* $\mathbf{L}_1$ *and* $\mathbf{L}_2$*, then it is also the underlying function for a morphism* $f : \mathsf{BPPO}(\mathbf{L}_1) \to \mathsf{BPPO}(\mathbf{L}_2)$.

PROOF: We begin by establishing that the induced order relation $\leq$ is indeed a partial order. It follows immediately from (gbwpl-iv) that it is reflexive. To show antisymmetry and transitivity, first observe the following. Suppose that $a \leq b$. If there is an $S \subseteq_f L$ such that $\bigvee S \downarrow$, $b = \bigvee S$, and $a \in S$, then $\bigvee\{a, b\} = b$ by a simple application of the generalized associativity laws. On the other hand, if there is an $S \subseteq_f L$ such that $\bigwedge S \downarrow$, $a = \bigwedge S$, and $b \in S$, then $\bigwedge\{a, b\} = a$, also by a simple application of the generalized associativity laws. But in this latter case $\bigvee\{a, b\} = \bigvee\{a, \bigwedge S\} = a$, by the generalized absorption identities. Thus, in either case $\bigvee\{a, b\} = b$.

To show antisymmetry, it suffices to observe that if $a \leq b$ and $b \leq a$, then both $\bigvee\{a, b\} = a$ and $\bigvee\{a, b\} = b$ hold, whence $a = b$. To show transitivity, assume that $a \leq b$ and $b \leq c$. Then we must have that $\bigvee\{a, b\} = b$ and $\bigvee\{b, c\} = c$. A simple application of the generalized associativity laws yields that $\bigvee\{a, b, c\} = c$, whence $a \leq c$, by the definition of $\leq$.

To complete the proof, we must prove that generalized joins are lub's, and generalized meets are glb's. So, suppose that S is a finite subset of L with $a = \bigvee S$, and let b be any upper bound for S. Then $s \leq b$ for all $b \in S$. Now, by the generalized associativity laws, we have $\bigvee\{b, a\} = \bigvee\{b, \bigvee S\} = \bigvee\{\bigvee\{b, s\} \mid s \in S\} = \bigvee S = a$. Hence $b = a$, and so $a = \mathrm{lub(S)}$. The proof for glb's is dual. This establishes condition (bppo-iv). To establish (bppo-v), let $S \subseteq_f L$, and let $a \in S$. Then, by the generalized absorption identities, $\bigwedge\{a, \bigvee S\} = a$. So, in particular, if $a = \mathrm{lub(S)}$, then we have (bppo-v), with the condition for glb's dual.

The fact that a morphism for GBWPL's is also a morphism for BPPO's is immediate. □

The other direction is much easier.

1.2.6 Proposition *Let* $\mathbf{L} = (L, \leq, \top, \bot, \bigvee, \bigwedge)$ *be a BPPO. Then* $\mathsf{GBWPL}(\mathbf{L})$ *is a GBWPL. Furthermore, if* $f : B_1 \to B_2$ *is the underlying function of a morphism between two GBWPL's* $\mathbf{L}_1$ *and* $\mathbf{L}_2$*, then it is also the underlying function for a morphism* $f : \mathsf{GBWPL}(\mathbf{L}_1) \to \mathsf{GBWPL}(\mathbf{L}_2)$.

PROOF: The only non-obvious conditions to prove are those of (gbwpl-vi), the generalized absorption identities. But if $S \subseteq_f B$ and $a \in S$, then $\bigwedge\{\{a\}, \bigvee S\} = \bigwedge\{\{a\}, \mathrm{lub(S)}\} = \mathrm{a}$, since $a \leq \mathrm{lub(S)}$, and so is the glb of $\{a, \bigvee S\}$. The other absorption condition is dual. □

We can now conclude formally that the notions of BPPO and of GBWPL are just two ways of talking about the same thing.

1.2.7 Proposition — equivalence of BPPO's and GBWPL's *The functors* BPPO *and* GBWPL *are mutually inverse. More specifically, we have the following.*

(a) *For any BPPO* $\mathbf{P}$*,* $\mathsf{BPPO}(\mathsf{GBWPL}(\mathbf{P})) = \mathbf{P}$*.*

(b) *For any GBWPL* $\mathbf{L}$*,* $\mathsf{GBWPL}(\mathsf{BPPO}(\mathbf{L})) = \mathbf{L}$*.*

PROOF: Most of the proof is clear from the constructions of the previous two propositions. The only question might be whether the transitions BPPO → GBWPL → BPPO might add pairs to the order relation. The key to observing that this cannot be the case is to note that the order relation of a BPPO is in fact already recaptured in the partial operations $\bigvee$ and $\bigwedge$. Indeed, $a \leq b$ iff $\bigvee\{a, b\} = b$, as we can readily conclude from (bppo-v). □

1.2.8 Natural interpretation of a lattice as a BPPO and as a GBWPL Often, in what follows, we will speak of a bounded lattice as being a BPPO or a GBWPL. In this sense, a lattice will simply be a BPPO or GBWPL in which the generalized join and generalized meet functions are total. Thus, for example, if $S = \{s_1, s_2, \ldots, s_n\}$, then $\bigvee S = s_1 \vee s_2 \vee .. \vee s_n$. That the axioms of a GBWPL are sufficient to guarantee that we obtain a bounded lattice whenever $\bigvee$ and $\bigwedge$ are total functions follows from an elementary axiomatization of a lattice. (See, *e.g.*, [Gra78, pp. 4–5].) Thus, whenever both $\bigvee$ and $\bigwedge$ are total functions, we shall speak of the corresponding BPPO or GBWPL as a lattice, with the obvious translation being made. Formally, we think of **BdLat**, as well as each of the distributive extension categories (1.1.5), as subcategories of **BPPO**.

1.3 Extensions of BPPO's to Lattices

We now turn to the key question of this section — the extension of BPPO's to bounded lattices of varying sorts. For consistency, we work with the problem of extending BPPO's; although, of course, in the light of the results above, we could equally well work with extending GBWPL's.

1.3.1 Definitions of Forms of Extensions Note that **BdLat** and any of the distributive extension categories may be naturally regarded as a subcategory of **BPPO**. For each such category $\mathcal{C}$, we have a natural forgetful functor which is the "identity" on objects — it forgets that the structure is more than just a BPPO. It is also the identity on morphisms, in the sense that the underlying functions are the same. The *extension problem*, simply put, is to find natural inverses to these functors. That is, to find natural ways to extend BPPO's to bounded lattices, bounded distributive lattices, and to Boolean lattices. Unfortunately, the situation is somewhat complex. First of all, there is not a unique notion of extension, but rather several distinct notions, each with its advantages and disadvantages. Secondly, in the cases of extensions which are required to be distributive, such extensions need not always exist. Indeed, the main topic of Section 2 of this report is to establish the computational complexity of the decision problem for the existence of such extensions. We must therefore address the problem on a case-by-case basis. Nonetheless, it is conceptually simpler to collect the basic definitions about extension into a common framework, and that is what we do here.

Let $\mathbf{P} = (P, \leq, \top, \bot, \bigvee, \bigwedge)$ be a BPPO, and let $\mathcal{C}$ be **BdLat** or one of the distributive extension categories. All of the types of extension have the form $(\mathbf{L}, \eta)$, in which $\mathbf{L}$ is an object of $\mathcal{C}$ and $\eta : \mathbf{P} \to \mathbf{L}$ is a BPPO-morphism. (Recall that $\mathcal{C}$ is always a subcategory of **BPPO**, so such a morphism is well-defined.) Let us call such a pair an *extension pair* for $\mathbf{P}$ to $\mathcal{C}$. The particulars for the special cases are as follows.

(a) The extension pair $(\mathbf{L}, \eta)$ is an *injective extension* of $\mathbf{P}$ to $\mathcal{C}$ if η is injective. More generally, for Q a set of pairs of distinct elements of P, $(\mathbf{L}, \eta)$ is called a *Q-injective extension* if for each pair $\{x, y\} \in Q$, $\eta(x) \neq \eta(y)$.

(b) The extension pair $(\mathbf{L}, \eta)$ is an *embedding extension* if η is an embedding. By an *embedding*, we mean that it has the property that it is injective and whenever $S \subseteq_f P$ is such that $\bigvee \eta(S) \downarrow$ (resp. $\bigwedge \eta(S) \downarrow$), then $\bigvee S \downarrow$ (resp. $\bigwedge S \downarrow$), and $\bigvee \eta(S) = \eta(\bigvee S)$ (resp. $\bigwedge \eta(S) = \eta(\bigwedge S)$).

(c) The extension pair $(\mathbf{L}, \eta)$ is *universal* if for every object $\mathbf{M}$ of $\mathcal{C}$ and every BPPO morphism $f : \mathbf{P} \to \mathbf{M}$, there is a unique $\mathcal{C}$ morphism g such that the diagram in Figure 1.1 commutes.

$$\begin{tikzcd} \mathbf{P} \arrow[r, "\eta"] \arrow[dr, "f"'] & \mathbf{L} \arrow[d, dashed, "g"] \\ & \mathbf{M} \end{tikzcd}$$

Figure 1.1: Universal extension.

In some sense, a universal extension is the most natural kind to seek, because it imposes the least amount of constraints, since any other extension is the homomorphic image of it. This idea arises in many mathematical settings. For a more general discussion of the merits of universal constructions, consult [Wal91, Ch. 5, Sec. 6] or [HS73, Sec. 26].

There are a few general facts about universal extension pairs which are worth recording at this point.

1.3.2 Proposition *Let $\mathbf{P}$ be a BPPO, and let $\mathcal{C}$ be the category* **BdLat** *or one of the distributive extension categories.*

(a) *A universal extension pair for $\mathbf{P}$ to $\mathcal{C}$, if it exists, is unique up to isomorphism.*

(b) *If $\mathbf{P}$ has an injective (resp. embedding) extension to $\mathcal{C}$, then a universal extension pair, if it exists, is also injective (resp. embedding).*

PROOF: Part (a) is a standard result from category theory; see, for example, [HS73, 26.7]. To prove (b), let $(\mathbf{M}, f)$ be an injective extension of $\mathbf{P}$, and let $(\mathbf{L}, \eta)$ be a universal extension pair. Then there is a morphism $g : \mathbf{L} \to \mathbf{M}$ which renders the diagram of Figure 1.1 commutative. But then η must be injective; again this is a special case of a standard categorical result [HS73, 6.1]. Finally, suppose that $(\mathbf{M}, f)$ of Figure 1.1 is an embedding extension, with $(\mathbf{L}, \eta)$ a universal extension. Let $S \subseteq_f P$ and $a \in P$ with $\bigvee \eta(S) = \eta(a)$. Then, since g is a morphism, we have that $\bigvee g(\eta(S)) = g(\eta(a))$. But $g \circ \eta = f$, thus $\bigvee h(S) = h(a)$, and since h is an embedding, $\bigvee S\!\downarrow$ with $\bigvee S = a$. The proof for the case of $\bigwedge$ is similar. Thus η is an embedding, as was to be proved. $\square$

1.3.3 Remark on injectivity of universal extensions In many mathematical domains, universal (or *free*) constructions have an injective "η". However, this is not necessarily the case with extensions of BPPO's to distributive extension categories. In 1.4.15, examples of universal extensions $(\mathbf{L}\eta)$ for BPPO's are presented for which η is not injective.

1.3.4 Remark on embedding We should emphasize that for lattices, as we noted in 1.1.2, injective morphisms are always embeddings. That is, if $f : \mathbf{L}_1 \to \mathbf{L}_2$ is a morphism of lattices, with $S \subseteq L_1$ and $a \in L_1$ such that $\bigvee\{f(s) \mid s \in S\} = f(a)$ (resp. $\bigwedge\{f(s) \mid s \in S\} = f(a)$) holds in $\mathbf{L}_2$, then $\bigvee S = a$ (resp. $(\bigwedge S = a)$ holds in $\mathbf{L}_1$. However, this is not true for partial structures. For example, if we have that $\bigvee S = \top$ in a BPPO, then in any extension to a lattice we must also have that $\bigvee T = \top$ for any finite T with the property that $S \subseteq T$. However, the axioms of a BPPO do not force $\bigvee T = \top$ to hold for every such superset T of S. For a less trivial example, see [Gra78, pp. 41–42]. Thus, for the notions of extension identified above, there may be a distinction between injective and embedding extensions. This issue is examined more closely in 1.3.6 and 1.3.7.

1.3.5 Ideals of a BPPO Let $\mathbf{P} = (P, \leq, \top, \bot, \bigvee, \bigwedge)$ be a BPPO. A set $I \subseteq P$ is called an *ideal* of $\mathbf{P}$ if it satisfies the following conditions:

(ideal-i) $\bot \in I$.

(ideal-ii) $a \in I$ and $b \leq a$ implies $b \in I$.

(ideal-iii) $S \subseteq I$ and $\bigvee S\!\downarrow$ implies $\bigvee S \in I$.

We let $\mathsf{Ideals}(\mathbf{P})$ denote the set of all ideals of $\mathbf{P}$. Observe that if $\mathbf{P}$ is a bounded lattice, then an ideal in this sense corresponds to a lattice ideal [Gra78, p. 17]. As in the case of lattices ([Gra78, Ch. I, Sec. 3, Cor. 2]), the set of ideals of a BPPO admits a natural structure as a lattice. Specifically, we define the bounded lattice $\mathbf{Ideals}(\mathbf{P}) = (\mathsf{Ideals}(\mathbf{P}), P, \emptyset, \bar{\cup}, \cap)$ associated with the BPPO $\mathbf{P}$ as follows. It is trivial to verify that the intersection of any number (possibly infinite) of ideals is itself an ideal. Intersection of (two) ideals thus forms the meet operation in $\mathbf{Ideals}(\mathbf{P})$. The join of two ideals is the intersection of all of the ideals which contain their union. More precisely, for $I, J \in \mathsf{Ideals}(\mathbf{P})$, define $I\overline{\cup}J = \bigcap\{K \in \mathsf{Ideals}(P) \mid I \subseteq K \text{ and } J \subseteq K\}$.

Given a set $S \subseteq P$, the *ideal generated by* S is the intersection of all ideals containing S, and is denoted by $\mathsf{Ideal}(S)$, or by $\mathsf{Ideal}(S, \mathbf{P})$ if we need to make explicit which BPPO we are talking about. Since P itself is such an ideal, this intersection is nontrivial. As in the case of ideals of lattices, for $a \in P$ we call $\mathsf{Ideal}(\{a\})$ the *principal ideal* generated by a.

The dual concept is defined as follows. $I \subseteq S$ is a *dual ideal* if it satisfies the following conditions.

(dideal-i) $\top \in I$.

(dideal-ii) $a \in I$ and $a \leq b$ implies $b \in I$.

(dideal-iii) $S \subseteq I$ and $\bigwedge S\!\downarrow$ implies $\bigwedge S \in I$.

We let $\mathsf{DualIdeals}(\mathbf{P})$ denote the set of all ideals of $\mathbf{P}$. The dual ideals form a bounded lattice $\mathbf{DualIdeals}(\mathbf{P}) = (\mathsf{DualIdeals}(\mathbf{P}), P, \emptyset, \overline{\cup}^d, \cap)$ in a completely analogous fashion, with $I\overline{\cup}^d J = \bigcap\{K \in \mathsf{DualIdeals}(\mathbf{P}) \mid I \subseteq K \text{ and } J \subseteq K\}$. The dual ideal generated by the set S is denoted by $\mathsf{DualIdeal}(S)$, or $\mathsf{DualIdeal}(S, \mathbf{P})$.

Given the BPPO $\mathbf{P} = (P, \leq, \top, \bot, \bigvee, \bigwedge)$, we have two natural injections. The first, $\mathsf{LowerExt}(\mathbf{P}) : \mathbf{P} \to \mathbf{Ideals}(\mathbf{P})$, called the *lower extension* of $\mathbf{P}$, is defined by $a \mapsto \mathsf{Ideal}(\{a\})$. The second, $\mathsf{UpperExt}(\mathbf{P}) : \mathbf{P} \to \mathsf{Dual}(\mathbf{DualIdeals}(\mathbf{P}))$, called the *upper extension* of $\mathbf{P}$, is defined by $a \mapsto \mathsf{DualIdeal}(\{a\})$. It is trivial to verify that each of these functions is injective and a BPPO morphism. We clearly have the following.

1.3.6 Theorem — extension of a BPPO to a bounded lattice *Any BPPO has an injective extension pair to* **BdLat**.

PROOF: $(\mathbf{Ideals}(\mathbf{P}), \mathsf{LowerExt}\mathbf{P}))$ and $(\mathsf{Dual}(\mathbf{DualIdeals}(\mathbf{P})), \mathsf{UpperExt}(\mathbf{P}))$ are each such extensions. $\square$

Thus, the extension problem for BPPO's to lattices is, in some sense, trivial. We shall show in the next section, however, that it is far from trivial if we demand that the extension be distributive. First, though, let us take a closer look at the question of when such an extension is an embedding. We start with a generalization of the notion of a generalized bounded partial lattice, which is a direct generalization of the concept of *partial lattice* identified in [Gra78, Ch. I., Sec. 5, Def. 12].

1.3.7 Generalized Bounded Partial Lattices A GBWPL is a *Generalized Bounded Partial Lattice* (GBPL) if it has an *embedding* extension into a bounded lattice.

A GBWPL (or, equivalently, a BPPO) $\mathbf{P}$ satisfies the *lower ideal condition* if for any $S \subseteq_f P$ and any $a \in S$, if $\mathsf{Ideal}(S, \mathbf{P}) = \mathsf{Ideal}(\{a\}, \mathbf{P})$ then $\bigvee S\downarrow$ and $\bigvee S = a$. Dually, it satisfies the *upper ideal condition* if for any $S \subseteq_f P$ and $a \in P$, if $\mathsf{DualIdeal}(S, \mathbf{P}) = \mathsf{DualIdeal}(\{a\}, \mathbf{P})$ then $\bigwedge S\!\downarrow$ and $\bigwedge S = a$.

1.3.8 Theorem — characterization of GBPL's *Let* $\mathbf{P}$ *be a GBWPL. Then* $\mathbf{P}$ *is a GBPL iff it satisfies both the lower and the upper ideal conditions.*

PROOF: The proof is a generalization of that given in [Gra78, Ch. I, Sec. 5, Thm. 20], which is attributed to Funayama. First of all, assume that $\mathbf{P}$ is a GBPL, and let $(\mathbf{L}, \eta)$ be an embedding extension of $\mathbf{P}$. Let $S \subseteq_f P$ and $a \in P$ be such that $\mathsf{Ideal}(S, \mathbf{P}) = \mathsf{Ideal}(\{a\}, \mathbf{P})$. Then $\eta(\mathsf{Ideal}(\{a\}, \mathbf{P})) = \eta(\mathsf{Ideal}(S, P)) \subseteq \mathsf{Ideal}(\eta(S), \mathbf{L}) \cap \eta(P) = \mathsf{Ideal}(\bigvee \eta(S), \mathbf{L}) \cap \eta(P)$. Thus, we have that $\eta(a) \leq \bigvee \eta(S)$ in $\mathbf{L}$. On the other hand, we also have that $\eta(s) \leq \eta(a)$ for each $s \in S$, just by the assumed conditions on the ideals, and so we must have that $\eta(a) = \bigvee \eta(S)$. But since η is an embedding, this means that $\bigvee S \downarrow$, with $a = \bigvee S$. Proving that the upper ideal condition holds is dual.

Conversely, assume that $\mathbf{P}$ satisfies both the lower and upper ideal conditions. Let $\mathbf{L} = \mathbf{Ideals}(\mathbf{P}) \times \mathsf{Dual}(\mathbf{DualIdeals}(\mathbf{P}))$, and define the function $\eta : \mathbf{P} \to \mathbf{L}$ by $a \mapsto (\mathsf{Ideal}(\{a\}), \mathsf{DualIdeal}(\{a\}))$. We claim that the pair $(\mathbf{L}, \eta)$ is an embedding extension of $\mathbf{P}$. It follows immediately from 1.3.5 that η is an injective morphism. We need to show that it is an embedding. Let $S \subseteq_f P$ and $a \in P$ be such that $\eta(a) = \bigvee \eta(S)$. Then, by the lower ideal condition, we have that $\bigvee S\downarrow$ and $a = \bigvee S$. Similarly, if we have $T \subseteq_f P$ and $b \in P$ with $\eta(a) = \bigwedge \eta(T)$, then we may use the upper ideal condition to establish that $b = \bigwedge T$. Thus η is an embedding, as was to be shown. □

1.3.9 Universal Extensions The question of universal extensions of arbitrary BPPO's is a rather complex one, and we will not address it here. The interested reader may wish to look at [Gra78, Ch. I, Sec. 5] or [Bir67, Ch. 6, Sec. 8] for more information. From a computational point of view, one of the problems is that a free (universal) lattice over a finite BPPO may be infinite. Fortunately, the problem of existence of free *distributive* lattices is somewhat more accessible, and it is to this important problem that we now turn.

1.4 Extensions of BPPO's to Distributive and Boolean Lattices

Extensions of BPPO's to distributive lattices are closely related to forms of binary decompositions of the lattice — that is, to the set of morphisms from the BPPO into the two-element lattice $\mathbf{2}$. We now turn to a development of these ideas.

1.4.1 Ideal binary decompositions Let $\mathbf{P}$ be a BPPO. An *ideal binary decomposition* of $\mathbf{P}$ is a pair (I, J) in which $I \in \mathsf{Ideals}(\mathbf{P})$, $J \in \mathsf{DualIdeals}(\mathbf{P})$, and $\{I, J\}$ forms a partition of the underlying set P, *i.e.*, $I \cup J = P$ and $I \cap J = \emptyset$.

1.4.2 Lemma — characterization of ideal binary decompositions *Let* $\mathbf{P}$ *be a BPPO, and let* $f : P \to \{\bot, \top\}$*. Then* f *is the underlying function of a BPPO morphism* $\mathbf{P} \to \mathbf{2}$ *iff* $(f^{-1}(\{\bot\}), f^{-1}(\{\top\}))$ *is an ideal binary decomposition of* $\mathbf{P}$*.*

PROOF: Assume that $f : \mathbf{P} \to \mathbf{2}$ is a BPPO morphism. Suppose that $S \subseteq_f P$ is such that $\bigvee S\downarrow$, with $S \subseteq f^{-1}(\bot)$. Then, $\bot = \bigvee f(S) = f(\bigvee S)$, so $\bigvee S \in f^{-1}(\bot)$. Hence $f^{-1}(\bot) \in \mathsf{Ideals}(\mathbf{P})$. Similarly, $f^{-1}(\top) \in \mathsf{DualIdeals}(\mathbf{P})$. Since $\{f^{-1}(\{\bot\}), f^{-1}(\{\top\})\}$ is clearly a partition of P, we have an ideal binary decomposition.

Conversely, suppose that $f : P \to \{\bot, \top\}$ is a function with the property that $(f^{-1}(\{\bot\}), f^{-1}(\{\top\}))$ is an ideal binary decomposition of $\mathbf{P}$. Then if $S \subseteq_f P$ is such that $\bigvee S\downarrow$ and for each $s \in S$, $f(s) = \bot$, then $f(\bigvee S) = \bot$ as well, since $f^{-1}(\{\bot\}) \in \mathsf{Ideals}(\mathbf{P})$. Similarly, if $S \subseteq_f P$ is such that $\bigwedge S\downarrow$ and for each $s \in S$, $f(s) = \top$, then $f(\bigwedge S) = \top$ as well, since $f^{-1}(\{\top\}) \in \mathsf{DualIdeals}(\mathbf{P})$. Hence f is a morphism, as required. □

This correspondence between ideal binary decompositions and morphisms into $\mathbf{2}$ is so central to our results that we need to make the identification explicit. This is done as follows.

1.4.3 Morphisms into the lattice 2 Let $\mathbf{P}$ be a BPPO, and let (I, J) be an ideal binary decomposition of $\mathbf{P}$. Define the morphism $\mathsf{Ifn}_{(I,J)} : \mathbf{P} \to \mathbf{2}$ by

$$\mathsf{Ifn}_{(I,J)} = \begin{cases} \bot & \text{if } p \in I; \\ \top & \text{if } p \in J. \end{cases}$$

Let $\mathsf{IdBinDecomp}(\mathbf{P}) = \{\mathsf{Ifn}_{(I,J)} \mid (I, J) \text{ is an ideal binary decomposition of } \mathbf{P}\}$.

For bounded bit-vector lattices, we employ an additional notation. Let $\mathbf{L}$ be a bounded bit-vector lattice over the set S, and let $s \in S$. Define the *s-projection* $\pi_s : L \to \{\bot, \top\}$ by

$$\pi_s(v) = \begin{cases} \bot & \text{if } v(s) = 0; \\ \top & \text{if } v(s) = 1. \end{cases}$$

1.4.4 Nonredundancy and independence Let $\mathbf{L}$ be a bounded bit-vector lattice over the set S. We say that the pair $s_1, s_2 \in S$ is *logically identical* if for every $f \in L$, $f(s_1) = f(s_2)$. In a situation with s_1 and s_2 logically identical, one of them may be removed without changing the lattice, up to isomorphism. $\mathbf{L}$ is called *nonredundant* if it does not contain any logically identical pairs. Clearly, we can always render a bounded bit-vector lattice nonredundant, provided the axiom of choice is assumed. For the most part, it will be convenient for us to work with nonredundant representations.

Let $v \in L$. For any $R \subseteq S$, we define $\mathsf{BitComp}(v, R)$ to be the function defined by

$$\mathsf{BitComp}(v, R) = \begin{cases} 1 - v(x) & \text{if } x \in R; \\ v(x) & \text{otherwise.} \end{cases}$$

Now let $\mathbf{M}$ be any bounded lattice, and let $f : \mathbf{L} \to \mathbf{M}$ be a morphism. For $s \in S$, we say that f is *independent* of s if for any bit vector $v \in L$, we have that whenever $\mathsf{BitComp}(v, \{s\}) \in L$, then $f(v) = f(\mathsf{BitComp}(v, \{s\}))$. Otherwise, we say that f is *dependent* upon s.

The following critical observation tells us that the only morphisms from a bounded bit-vector lattice into **2** are the projections of a single "bit."

1.4.5 Proposition — characterization of morphisms into the lattice 2 *Let S be a set, let $\mathbf{L}$ be a bounded bit-vector lattice over S. Then $f : L \to \{\bot, \top\}$ is the underlying function of a morphism $\mathbf{L} \to \mathbf{2}$ iff $f = \pi_s$ for some $s \in S$.*

PROOF: It is trivial to verify that each of the π_s's are morphisms; we need to establish the opposite direction. So, let $f : \mathbf{L} \to \mathbf{2}$ be a morphism. Without loss of generality, assume that S does not contain any distinct logically identical pairs. Suppose that there are two distinct elements $s_1, s_2 \in S$ which f is dependent upon. Then there are vectors $v_{11}, v_{12}, v_{21}, v_{22} \in L$ such that $v_{12} = \mathsf{BitComp}(v_{11}, \{s_1\})$, $v_{22} = \mathsf{BitComp}(v_{21}, \{s_2\})$, $v_{11}(s_1) = 1$, $v_{21}(s_2) = 1$, $f(v_{11}) = f(v_{21}) = 1$ and $f(v_{12}) = f(v_{22}) = 0$. Now, since f is a morphism, we must have that $f(v_{11} \wedge v_{21}) = 1$ and $f(v_{12} \vee v_{22}) = 0$. But it follows from the above definitions that $v_{11}(s_1) \wedge v_{21}(s_1) = v_{12}(s_1) \vee v_{22}(s_1)$. For if not, then we must have that $v_{11}(s_1) = v_{21}(s_1) = 1$ and $v_{12}(s_1) = v_{22}(s_1) = 0$. But this cannot happen, since $v_{21}(s_1) = v_{22}(s_1)$. Similarly, $v_{11}(s_2) \wedge v_{21}(s_2) = v_{12}(s_2) \vee v_{22}(s_2)$. Additionally, v_{11} and v_{12} agree on all positions other than s_1, and v_{21} and v_{22} agree on all positions other than s_2. Thus $v_{11}(s) \wedge v_{21}(s) \leq v_{12}(s) \vee v_{22}(s)$ for all s other than s_1 or s_2. But we have just shown that $v_{11}(s) \wedge v_{21}(s) = v_{12}(s) \vee v_{22}(s)$ for $s = s_1$ or s_2. Thus $v_{11}(s) \wedge v_{21}(s) \leq v_{12}(s) \vee v_{22}(s)$ for all $s \in S$; *i.e.*, $v_{11} \wedge v_{21} \leq v_{12} \vee v_{22}$. Since a lattice morphism is monotonic, this means that $f(v_{11} \wedge v_{21}) \leq f(v_{12} \vee v_{22})$, a contradiction. Hence f can be dependent upon only one of s_1 and s_2. From this it follows that it must be a projection. □

The ideal binary decompositions of a bounded bit-vector lattice are then in bijective correspondence with the projections of "bits."

1.4.6 Proposition — decompositions of bounded distributive lattices *Let S be a set, and let $\mathbf{L}$ be a bounded bit-vector lattice over S. Let $I, J \subseteq L$. Then (I, J) is an ideal binary decomposition of $\mathbf{L}$ iff there is an $s \in S$ such that $I = \{x \in L \mid \pi_s(x) = \bot\}$ and $J = \{x \in L \mid \pi_s(x) = \top\}$.*

PROOF: Follows from 1.4.2 and 1.4.5. □

We are now ready to state our first extension result to distributive extension categories. Namely, the existence of such an extension depends entirely upon the existence of an ideal binary decomposition.

1.4.7 Theorem — existence of extension pairs *Let $\mathbf{P}$ be a BPPO. Then, for any of the distributive extension categories, $\mathbf{P}$ has an extension pair iff it has an ideal binary decomposition.*

PROOF: If $\mathbf{P}$ has an ideal binary decomposition, then by 1.4.2 it has a morphism $f : \mathbf{P} \to \mathbf{2}$, which renders $(\mathbf{2}, f)$ an extension pair. Conversely, let $(\mathbf{L}, \eta)$ be an extension pair. Since $\mathbf{L}$ is distributive, by 1.1.4(a) we know that it is isomorphic to a bounded bit-vector lattice $\mathbf{M}$ over a set S. Then, by 1.4.6, we know that for any $s \in S$ there is a projection $\pi_s : \mathbf{M} \to \mathbf{2}$. Then, letting $\iota : \mathbf{L} \to \mathbf{M}$ denote the isomorphism between $\mathbf{L}$ and $\mathbf{M}$, we have that $\pi_s \circ \iota \circ \eta : \mathbf{P} \to \mathbf{2}$. So, finally, applying 1.4.2, we get an ideal binary decomposition. □

1.4.8 Examples Let us illustrate these ideas with an example. Let $\mathbf{P}_1$ be the lattice (*qua* BPPO) represented by the diamond of Figure 0.3. It is easy to see that it has no ideal binary decompositions at all. Indeed, let (I, J) be such a decomposition. Then we must have that $e \in I$. Now if any pair $\{x, y\}$ of elements from $\{b, c, d\}$ is in I, then we must also have that $a \in I$ as well, since the constraint $x \vee y = a$ holds. But if both $a, e \in I$, then the decomposition is trivial; *i.e.*, $J = \emptyset$. Dually, if the pair $\{x, y\}$ is in J, then we must also have that $e \in J$ as well, since the constraint $x \wedge y = e$ holds. Since some pair of elements from $\{b, c, d\}$ must be in either I or else in J, it follows that no ideal binary decomposition can exist.

On the other hand, let $\mathbf{P}_2$ be the lattice (*qua* BPPO) represented by the pentagon of Figure 0.2. Then we have two ideal binary decompositions, $(\{a, b\}, \{c, d, e\})$ and $(\{a, c, d\}, \{b, e\})$.

To get injective decompositions, we essentially need to ensure that there are enough ideal binary decompositions, where "enough" means a sufficient number to ensure that for any two distinct elements, there is a decompositions which sends those elements to distinct members of $\{\bot, \top\}$.

1.4.9 Separating families of decompositions Let $\mathbf{P}$ be a BPPO, and let $\{x, y\} \subseteq P$. We say that x and y are *separable* if there is a morphism $f : \mathbf{P} \to \mathbf{2}$ with $f(x) \neq f(y)$. In this case, we call f a *separator* for $\{x, y\}$. We say that $\mathbf{P}$ is *somewhere separable* if there is a pair $\{x, y\} \subseteq P$ which is separable. It is clear that somewhere separability is equivalent to the separability of $\{\bot, \top\}$, since any morphism must separate $\bot$ from $\top$.

Now let Δ be a family of BPPO morphisms, each of the form $\mathbf{P} \to \mathbf{2}$, and let Q be a set of pairs of elements of P. The family Δ is called a *separator* for Q (or a *Q-separator*) if it contains a separator for each pair in Q. It is called a *(full) separator* for $\mathbf{P}$ if it contains a separator for every pair $\{x, y\} \subseteq P$ with $x \neq y$. In the case that such a Δ exists, we say that $\mathbf{P}$ is *fully separable*.

1.4.10 Theorem — existence of extensions *Let $\mathbf{P}$ be a BPPO, and let $\mathcal{C}$ be one of the distributive extension categories.*

(a) $\mathbf{P}$ *has an extension pair into $\mathcal{C}$ iff it is somewhere separable.*

(b) *For Q a set of pairs of elements of P, $\mathbf{P}$ has a Q-injective extension pair into $\mathcal{C}$ iff Q has a separator.*

(c) $\mathbf{P}$ *has an injective extension into $\mathcal{C}$ iff it is fully separable.*

PROOF: Part (a) is just a restatement of 1.4.7 in terms of separators. To show part (b), let Q be a set of pairs of elements of P, and $(\mathbf{L}, \eta)$ be a Q-injective extension into $\mathcal{C}$. Without loss of generality, we may take $\mathbf{L}$ to be a bounded bit-vector lattice.

Let S be the index set over which this lattice is taken. It is clear that $\{\pi_s \mid s \in S\}$ is fully separating for $\mathbf{L}$. But then, since η is injective, it must be the case that $\{\pi_s \circ \eta \mid s \in S\}$ is separating for each pair in Q.

Conversely, suppose that Q has a separator Δ. Write this set as an indexed set: $\Delta = \{f_s : \mathbf{P} \to \mathbf{2} \mid s \in S\}$. Let $\mathbf{L} = \prod_{s \in S} 2$, and let $\eta : \mathbf{P} \to \mathbf{L}$ be given by $p \mapsto \langle f_s(p) \rangle_{s \in S}$. Then $\eta(x) \neq \eta(y)$ for any pair $\{x, y\} \in Q$, since $f_s(x) \neq f_s(y)$ for some $s \in S$. Thus $(\mathbf{L}, \eta)$ is Q-injective, as required.

Part (c) is a special case of (b), with Δ consisting of all pairs of distinct elements of P. $\square$

1.4.11 Examples We continue with the examples of 1.4.8. Since $\mathbf{P}_1$ has no ideal binary decompositions, it has no distributive extensions at all. Neither can $\mathbf{P}_2$ have an injective distributive extensions, since there is not an ideal binary decomposition which separates the pair $\{c, d\}$. In effect, what this says is that to make a pentagon distributive, we must collapse the two elements on the "long" path from the bottom to the top. In a diamond, we have no choice but to collapse the whole thing to a single element.

Finally, we turn to the issue of universal extensions. For the distributive extension category of perfect Boolean lattices, the universal extension uses the product of *all* extension pairs. For the other categories, it is a matter of restricting this product appropriately. The details follow.

1.4.12 Theorem — existence of universal extensions over PerfBoolLat *Let* $\mathbf{P}$ *be a BPPO. Then,* $\mathbf{P}$ *has a universal extension over* **PerfBoolLat** *iff it has an extension over* **PerfBoolLat**. *The universal extension is injective iff there is some injective extension over* **PerfBoolLat**. *The explicit construction is as follows. Let* $F = \mathsf{IdBinDecomp}(\mathbf{P})$, *and let* $\eta : \mathbf{P} \to \mathbf{2}^F$ *be defined by* $p \mapsto \langle f(p) \rangle_{f \in F}$. *Then* $(\mathbf{2}^F, \eta)$ *is a universal extension of* $\mathbf{P}$ *to a perfect Boolean lattice.*

PROOF: If $\mathbf{P}$ has no extension at all, then it certainly has no universal extension. Thus, let us assume that it has at least one extension to **PerfBoolLat**. In view of 1.4.7, this can happen iff F is nonempty. Now, let $\mathbf{M}$ be any perfect Boolean lattice, and let $g : \mathbf{P} \to \mathbf{M}$ be a BPPO morphism. Without loss of generality, we may take $\mathbf{M}$ to be $\mathbf{2}^S$ for some set S. Now, consider the diagram of in Figure 1.2.

We need to show that there is a unique morphism $\hat{g}$ which makes the left triangle commute. First of all, for $t \in T$, $\pi_t \circ g : \mathbf{P} \to \mathbf{2}$, and so must be a member of F. Hence there is some $f \in F$ such that $\pi_f \circ \eta = \pi_t \circ g$. Name this π_f as h_t. Now, since $\mathbf{2}^T$ is a product in **PerfBoolLat**, there is a unique morphism $\hat{g} : \mathbf{2}^F \to \mathbf{2}^T$ making the right triangle commute. But $\pi_t \circ \hat{g} \circ \eta = \pi_t \circ g$ for each $t \in T$. Thus, using the fact once again that $\mathbf{2}^T$ is a product, we have that $\hat{g} \circ \eta = g$. The uniqueness of $\hat{g}$ follows from the uniqueness of the fill-in for a product [HS73, 18.1]. Hence $(\mathbf{2}^F, \eta)$ is a universal extension of $\mathbf{P}$ over **PerfBoolLat**.

Now if $\mathbf{P}$ has an injective extension, then by 1.4.10(b) we know that it has a separating set of ideal binary decompositions. Thus, in particular, the set $\mathsf{IdBinDecomp}(\mathbf{P})$ must be separating. From this it follows directly that η is injective.

$$
\begin{tikzcd}
\mathbf{P} \arrow[r, "\eta"] \arrow[dr, "g"'] & \mathbf{2}^F \arrow[d, dashed, "\hat{g}"'] \arrow[dr, "\pi_f = h_t"] & \\
& \mathbf{2}^T \arrow[r, "\pi_t"'] & \mathbf{2}
\end{tikzcd}
$$

Figure 1.2:

□

1.4.13 Corollary — existence of universal extensions over distributive extension categories *Let* $\mathbf{P}$ *be a BPPO, and let* $\mathcal{C}$ *be a distributive extension category. Then* $\mathbf{P}$ *has a universal extension over one of the categories in* $\mathcal{C}$ *iff it has a universal extension over all of them. If one of these extensions is injective, then they all are. The explicit construction is as follows. Let* $(\mathbf{2}^F, \eta)$ *be the universal extension described in the above theorem. For the distributive extension category* **PerfBoolLat**, *we are done. Otherwise. let* $\mathbf{L}$ *be the* $\mathcal{C}$*-subalgebra of* $\mathbf{2}^F$ *(distributive lattice or Boolean lattice) generated by* $\eta(P)$, *and let* $\eta_{\mathcal{C}} : \mathbf{P} \to \mathbf{L}$ *be the morphism which is identical to* η, *except that its codomain is restricted to be* $\mathbf{L}$. *Then* $(\mathbf{L}, \eta_{\mathcal{C}})$ *is a universal extension of* $\mathbf{P}$ *to* $\mathcal{C}$.

Proof: If $\mathcal{C}$ is **PerfBoolLat**, then the result has already been proven in the above theorem. Otherwise, let $(\mathbf{2}^F, \eta)$ be a universal extension of $\mathbf{P}$ to **PerfBoolLat**, let $\mathbf{M}$ be a $\mathcal{C}$ object, and let $g : \mathbf{P} \to \mathbf{M}$ be a BPPO morphism. Consider the diagram of Figure 1.3.

$$
\begin{tikzcd}
\mathbf{P} \arrow[r, "\eta_{\mathcal{C}}"] \arrow[dr, "g"'] & \mathbf{L} \arrow[r, "\iota_{\mathbf{L}}"] \arrow[d, dashed, "\tilde{g}"] & \mathbf{2}^F \arrow[d, "\hat{g}"] \\
& \mathbf{M} \arrow[r, "\iota_{\mathbf{M}}"] & \mathbf{2}^T
\end{tikzcd}
$$

Figure 1.3:

Here $\mathbf{L}$ is the $\mathcal{C}$ object generated by $\eta(P)$ and $\iota_{\mathbf{L}}$ is the natural inclusion of $\mathbf{L}$ into $\mathbf{2}^F$. That is to say, if $\mathcal{C}$ is **BdDiLat**, then $\mathbf{L}$ is the smallest distributive sublattice containing $\eta(P)$. Explicitly, its underlying set is the intersection of the underlying sets of all distributive sublattices of $\mathbf{2}^F$ which contain $\eta(P)$. If $\mathcal{C}$ is **BoolLat**, then it is the smallest Boolean lattice containing $\eta(P)$.

Since $\mathbf{M}$ is a distributive lattice, in view of 1.1.4(c), we may embed it into a perfect Boolean lattice $\mathbf{2}^T$. We let $\iota_{\mathbf{M}}$ be such an embedding. From the previous theorem, we then have a morphism $\hat{g}$ such that the pentagon of the diagram commutes. To obtain the fill-in $\tilde{g}$, we simply use the appropriate restriction of $\hat{g}$. Now $\hat{g}(\iota_{\mathbf{L}}(L))$ is the underlying set of the smallest sublattice of $\mathbf{2}^T$ in $\mathcal{C}$ which contains $\hat{g}(\iota_{\mathbf{L}}(\eta_{\mathcal{C}}(P))) = \iota_{\mathbf{M}}(g(P)) = \iota_{\mathbf{M}}^{-1}(g(P))$. The image of M under $\iota_{\mathbf{M}}$ is also a sublattice of $\mathbf{2}^T$ which contains $\iota_{\mathbf{M}}(g(P))$, and so $\hat{g}(\iota_{\mathbf{L}}(L)) \subseteq \iota_{\mathbf{M}}(M)$. Thus, there is a unique well-defined fill-in $\tilde{g}$ defined by $x \mapsto \iota_{\mathbf{M}}^{-1}(\hat{g}(\iota_{\mathbf{M}})(x))$. It is easily verified that this fill-in is a morphism, thus completing the proof. □

1.4.14 Remark on the existence of complements The preceding corollary states that, from a fundamentals point of view, once we can get distributivity, a guarantee of the existence of various forms of complements comes for free. This is quite important because negation (*qua* complementation) is an important operation in many practical specifications. Once we know that we have a distributive extension of a particular kind, we know that we can extend it to one with complements. Thus, complements need not be studied separately.

1.4.15 Examples — universal extensions need not be injective Let us continue with the example $\mathbf{P}_2$ of 1.4.11. Over **PerfBoolLat**, the universal perfect Boolean lattice is $\mathbf{2}^2$. The morphism $\eta : \mathbf{P}_2 \to \mathbf{2}^2$ behaves as follows.

$$\begin{array}{rcl} a & \mapsto & (\top,\top) \\ b & \mapsto & (\top,\bot) \\ c & \mapsto & (\bot,\top) \\ d & \mapsto & (\bot,\top) \\ e & \mapsto & (\bot,\bot) \end{array}$$

Thus, c and d are identified in this universal pair; and, indeed, in every universal extension. Since η is surjective, the result is the same in the other distributive extension categories.

It is instructive to illustrate that we may construct a BPPO $\mathbf{P}_3$ with six elements which has only one decomposition, thus implying that a universal extension $(\mathbf{L}, \eta)$ over $\mathbf{P}_3$ must have that $\mathbf{L}$ is isomorphic to $\mathbf{2}$. Let $\mathbf{P}_3$ be the lattice (*qua* BPPO) illustrated in Figure 1.4. We know from the discussion of $\mathbf{P}_2$ in 1.4.8 that the diamond defined by the elements $\{a, b, c, d, e\}$ cannot have any ideal binary decomposition. Therefore, the only ideal binary decomposition of $\mathbf{P}_3$ is $(\{f\}, \{a, b, c, d, e\})$. Hence, by 1.4.13, we have that $(\mathbf{2}, \eta_3)$ is a universal extension of $\mathbf{P}_3$ for any of the distributive extension categories, with η_3 defined by $a, b, c, d, e \mapsto \top$ and $f \mapsto \bot$.

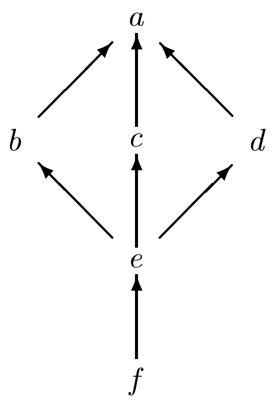

Figure 1.4: A bounded lattice whose universal distributive extension is **2**.

1.5 Atoms in BPPO's

Languages such as CUF allow types to be declared to be *constant*. A constant type is constrained so that it may not have any proper subtypes other than $\bot$. Within a lattice-like framework, the formalization of a constant type is that of an atom. In this section, we take a look at how such constraints may be viewed within our theoretical framework.

1.5.1 Basic definitions In a bounded lattice $\mathbf{L}$, an element $a \in L$ is an *atom* if for every $x \in L$, if $x \leq a$ then either $x = \bot$ or else $x = a$. Unfortunately, there does not appear to be a meaningful definition of an atom in a BPPO. We could say that, given a BPPO $\mathbf{P}$, $a \in P$ is an atom if for every extension $(\mathbf{L}, \eta)$ of $\mathbf{P}$, a is an atom of $\mathbf{L}$. However, it is always possible to find an extension of $\mathbf{P}$ in which a is not an atom, so this definition is vacuous. Rather, the best that we can do is to restrict attention to those extensions in which a is an atom. We thus make the following definition regarding atoms in the context of a particular extension. Let $\mathbf{P}$ be a BPPO, and let $S \subseteq P$. An extension $(\mathbf{L}, \eta)$ of $\mathbf{P}$ is *atomic for S* if for each $a \in S$, a is an atom in $\mathbf{L}$. We can then seek to characterize such atomic extensions. Unfortunately, sets of atomic extensions, even distributive ones, do not share all of the nice properties of sets of general distributive extensions. The following examples illustrates some of the principal problems.

1.5.2 Examples Let $\mathbf{P}$ be the BPPO with $P = \{\bot, \top, a, b\}$, and with no order on the symbols other than the required ones; *i.e.*, $\bot \leq a$, $\bot \leq b$, $a \leq \top$, and $b \leq \top$. The only generalized join and meet operations are the trivial ones implied by this order. Now suppose that we require a to be an atom; that is, suppose that $S = \{a\}$. Then it is easy to see that there can be no universal extension to **BdDiLat** which is atomic for S. For suppose that $(\mathbf{L}, \eta)$ is such an extension. Then there are two possibilities: either $\eta(b) \wedge \eta(a) = \eta(a)$ or else $\eta(b) \wedge \eta(a) = \bot$ in $\mathbf{L}$. Any other value for this meet would invalidate the requirement that a be an atom. Now let $(\mathbf{L}_1, \eta_1)$

and $(\mathbf{L}_2, \eta_2)$ be the simple extensions depicted on the left and right, respectively, in Figure 1.5.

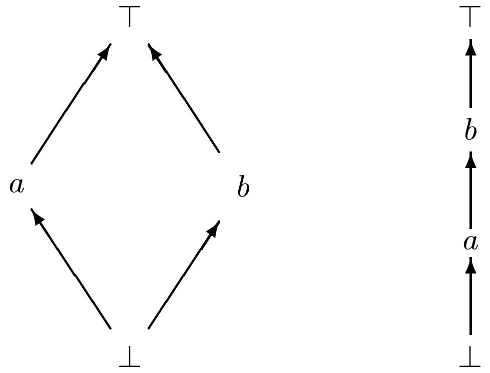

Figure 1.5: Two bounded distributive lattices in which a is an atom.

In these extensions, the associated morphism η_i is the identity on elements; the target lattice $\mathbf{L}_1$ or $\mathbf{L}_2$ just has more structure than $\mathbf{P}$. Now it is easy to see that no universal extension $(\mathbf{L}, \eta)$ in which a is an atom has the property that there are g_1 and g_2 such that the diagram of Figure 1.6 is commutative for both $i = 1$ and $i = 2$. Indeed, if $\eta(a) \wedge \eta(b) = \eta(a)$ in $\mathbf{L}$, then $g_1(\eta(a)) \wedge g_1(\eta(b)) = g_1(\eta(a))$ in $\mathbf{L}_1$, *i.e.*,

$$\begin{tikzcd}
\mathbf{P} \arrow[r, "\eta"] \arrow[dr, "\eta_i"'] & \mathbf{L} \arrow[d, dashed, "g_i"] \\
& \mathbf{L}_1
\end{tikzcd}$$

Figure 1.6: Diagram to support argument of the impossibility of universal extensions which preserve atoms.

$a \wedge b = a$, which fails. Similarly, if $\eta(a) \wedge \eta(b) = \bot$ in $\mathbf{L}$, then $g_2(\eta(a)) \wedge g_i(\eta(b)) = \bot$ in $\mathbf{L}_2$, *i.e.*, $a \wedge b = \bot$, which fails.

There is a further problem with atoms. Unlike equational properties such as being distributive or Boolean, the property of being an atom is not preserved under morphic image. Given any bounded lattice $\mathbf{L}_1$ and any atom $a \in L$, we may build a slightly larger lattice $\mathbf{L}_2$ in which a is not an atom, and an injective morphism $f : \mathbf{L}_1 \to \mathbf{L}_2$ which is the identity on L_1. Just pick a name $b \notin L_1$. Create $\mathbf{L}_2$ from $\mathbf{L}_1$ by having $L_2 = L_1 \cup \{b\}$. We put b "in between" a and $\bot$ by defining, in $\mathbf{L}_2$, $b \vee a = a$, $b \wedge a = b$, and, for any other $x \in L_1$, $b \vee x = a \vee x$ and $b \wedge x = a \wedge x$. It is easy to see that the lattice axioms are satisfied, and that a is no longer an atom. Thus, we can always "embed away" the property of being an atom.

From a practical point of view, the previous examples illustrate that, if we wish to specify that certain elements be atoms, then we cannot speak of *the* (up to isomorphism) natural extension of the BPPO $\mathbf{P}$ to a given context, nor can we characterize the property of an extension $(\mathbf{L}, \eta)$ of $\mathbf{P}$ by considering morphisms with domain $\mathbf{P}$, as we did in the previous section for other kinds of extensions. Rather, we must be more careful. We proceed as follows.

1.5.3 Atomic Extensions Let $\mathbf{P}$ be a BPPO, and let $\Delta \subseteq \{f \mid f : \mathbf{P} \to \mathbf{2}\}$. The *full natural extension* of $\mathbf{P}$ defined by Δ is defined to be $(\mathbf{2}^{\Delta}, \eta_{\Delta})$, with $\eta_{\Delta}(a) = \langle f(a) \rangle_{f \in \Delta}$. An extension $(\mathbf{L}, \eta)$ is a *natural extension of* $\mathbf{P}$ *with respect to* Δ if there is an injective morphism ι which preserves atoms (*i.e.*, maps atoms to atoms) such that the diagram in Figure 1.7 commutes. Note that this is a generalization of the

$$\begin{tikzcd} \mathbf{P} \arrow[r, "\eta"] \arrow[dr, "\eta_{\Delta}"'] & \mathbf{L} \arrow[d, dashed, "\iota"] \\ & \mathbf{2}^{\Delta} \end{tikzcd}$$

Figure 1.7: Diagram for 1.5.3.

constructions used in 1.4.12 and 1.4.13; in those articles we have the special case that $\Delta = \mathsf{IdBinDecomp}(\mathbf{P})$.

Now let $S \subseteq P \setminus \{\bot, \top\}$. We say that Δ *renders* S *atomic* if for every $a \in S$, there is exactly one $f \in \Delta$ such that $f(a) = \top$. (In other words, $f(a) = \bot$ for all $f \in \Delta$, save for possibly one.)

The following lemma is well-known, but it is crucial to our work, and easy to prove, so we repeat it here.

1.5.4 Lemma *Let* $\mathbf{L}$ *be a bounded distributive lattice. Then there is a perfect Boolean lattice* $\mathbf{B}$ *and an embedding* $\iota : \mathbf{L} \to \mathbf{B}$ *which preserves atoms.*

PROOF: Without loss of generality, we may take $\mathbf{L}$ to be a bounded bit-vector lattice. Let R be the index set for the bits of the bit vectors. For each atom $a \in L$, let R_a be the set of bits which are 1 in the representation of a. For a_1, a_2 distinct atoms, it is clear that $R_a \cap R_b = \emptyset$, since $a \wedge b = \bot$. Thus, we can replace, for each atom a, the set R_a with a single element, without changing $\mathbf{L}$, up to isomorphism. Once we do this, we are left with the situation that each atom of $\mathbf{L}$ is represented by a bit vector in which exactly one bit is nonzero. Let us call the new index set

R'. To complete the proof, we embed this modified $\mathbf{L}$ into the perfect lattice $\mathbf{2}^{R'}$. Since the atoms of $\mathbf{L}$ were represented by elements with a single nonzero bit, they continue to be atoms in $\mathbf{2}^{R'}$. □

Finally, we come to our main characterization of extensions to a BPPO which forces certain elements to be atoms.

1.5.5 Theorem — characterization of atoms *Let* $\mathbf{P}$ *be a BPPO, let* $S \subseteq P \setminus \{\bot, \top\}$*, and let* $(\mathbf{L}, \eta)$ *be an extension of* $\mathbf{P}$*. Then*

(a) *If* $\Delta \subseteq \{f \mid f : \mathbf{P} \to \mathbf{2}\}$ *renders* S *atomic and* $(\mathbf{L}, \eta)$ *is natural with respect to* Δ*, then* $(\mathbf{L}, \eta)$ *is atomic for* S*.*

(b) *If* $(\mathbf{L}, \eta)$ *is atomic for* S*, then there is a* $\Delta \subseteq \{f \mid f : \mathbf{P} \to \mathbf{2}\}$ *which renders* S *atomic.*

Proof: Part (a) is immediate; note that the atoms of $\mathbf{2}^{\Delta}$ are precisely the Δ-tuples in which exactly one entry is $\top$ and the rest are $\bot$. Thus, since Δ renders S atomic, it follows that each $\eta_{\Delta}(a)$ is an atom in $\mathbf{2}^{\Delta}$ for each $a \in S$. But ι of Figure 1.7 is an injection, and it is easy to see that in that case the inverse image of an atom must be an atom. Thus, $\eta(a)$ is an atom of $\mathbf{L}$ for each $a \in S$, and so $(\mathbf{L}, \eta)$ is atomic for S.

To show part (b), we simply invoke 1.5.4, and use the fact that any perfect Boolean lattice is of the form $\mathbf{2}^{A}$ for some set A. Thus, we can construct a morphism $h : \mathbf{P} \to \mathbf{2}^{A}$ which preserves atoms. Now, we define $\Delta = \{\pi_x \circ \eta \mid x \in A\}$. Since an element in $\mathbf{2}^{A}$ is an atom iff exactly one of its components is $\top$, it follows that Δ renders S atomic. □

1.5.6 Theorem — independence of the extension category *Let* $\mathbf{P}$ *be a BPPO, and let* $S \subseteq P \setminus \{\bot, \top\}$*. Then, if* $\mathbf{P}$ *has an extension (resp. injective extension) which is atomic for* S *to any one of the distributive extension categories, then it has such an extension to all of them.*

Proof: This follows immediately from 1.5.4, since that result shows that we can extend the extension to one in **PerfBoolLat**, which is a subcategory of all of the others. □

In summary, when dealing with specifications which require that certain types must be atomic, we must look for specific extensions which yields this result, rather than characterizing a universal extension which subsumes all others. In Section 2.5, we will return to the issue of atoms by addressing some of the computational issues.

2. Computational Complexity of Decision Problems

In this section, we turn to the problem of determining the computational complexity of deciding whether or not a BPPO has a distributive extension of one sort or another. For obvious reasons, we now restrict our attention to finite structures.

2.1 Determining Consistency of Prespecifications

A specification that a user of a system such as CUF supplies is not likely to be a BPPO. Rather, it will be some sort of "skeleton" that may be extended to a BPPO. To assess the complexity of the whole process of taking a specification and deciding whether or not it is extensible to a distributive lattice, we must therefore break the process into two distinct steps. First, we must determine if the specification is "well-formed," that is, if it represents a consistent specification of a BPPO. Second, if it does specify a BPPO, then we must decide whether or not it has the appropriate form of extension.

In this subsection, we address the question of determining whether a prespecification is well formed as a definition of a BPPO, and show that the problem of determining this is solvable in deterministic polynomial time. To begin, we give the definition of a prespecification.

2.1.1 BPPO prespecifications A *finite BPPO prespecification* is a pair $\mathbf{S} = (P, \mathbf{C})$ in which

(i) P is a finite set, called the set of *inner types*. We call $P \cup \{\bot, \top\}$ the set of *types* of $\mathbf{S}$, and denote it by $\mathsf{Aug}(P)$.

(ii) $\mathbf{C}$ is a finite set, called the set of *constraints*. Each constraint in $\mathbf{C}$ is an expression of one of the following three forms.

  (a) $(a < b)$, with $a, b \in \mathsf{Aug}(P)$ and $a \neq b$.

  (b) $(\bigvee S = a)$, with $a \in \mathsf{Aug}(P)$ and S a nonempty subset of P containing at least two distinct elements.

  (c) $(\bigwedge S = a)$, with $a \in \mathsf{Aug}(P)$ and S a nonempty subset of P containing at least two distinct elements.

It should be noted that a given constraint has many syntactic representations. For example, if $S = \{a_1, \ldots, a_k\}$, then $(\bigvee S = a)$ and $(\bigvee\{a_1, \ldots, a_k\} = a)$ denote the same constraint, even though they are not the same string. Also, we require that S in any rule of the form $(\bigvee S = a)$ or $(\bigwedge S = a)$ have at least two distinct elements, for otherwise the rule could only be consistent with a BPPO if $S = \{a\}$, which contributes nothing to the specification, since it follows from the axioms of a BPPO.

The *size* of a rule of the form $(a < b)$ is 2. The *size* of a rule of the form $(\bigvee S = a)$ or $(\bigwedge S = a)$ is $\mathsf{Card}(S) + 1$, where $\mathsf{Card}(S)$ denotes the cardinality of S. The *size* of $\mathbf{S}$ is $\mathsf{Card}(P)$ plus the sum of the sizes of its rules, and is denoted $\mathsf{Size}(\mathbf{S})$.

A prespecification is supposed to represent, in abstract form, the same level of specification language as, say, the CUF type definition language does. There are some differences in detail, although this will not affect the overall complexity or power. The appendix contains a more detailed description of this component of CUF, together with a comparison of the two.

For a prespecification to be meaningful, it must be consistent — that is, it must represent a BPPO. The next definition formalizes this idea.

2.1.2 Consistency of prespecifications Let $\mathbf{S} = (P, \mathbf{C})$ be a finite BPPO prespecification, and let $\mathbf{B} = (B, \leq, \top, \bot, \bigvee, \bigwedge)$ be a BPPO. Informally, $\mathbf{B}$ is consistent with $\mathbf{S}$ if it can be extended to S in a "natural" way. Formally, we say that $\mathbf{B}$ is *consistent* with $\mathbf{S}$ if it satisfies the following conditions.

(cs-i) $B = \mathsf{Aug}(P)$.

(cs-ii) $(a < b) \in \mathbf{C}$ implies that $a \leq b$ in the poset of $\mathbf{B}$.

(cs-iii) $(\bigvee S = a) \in \mathbf{C}$ implies that $\bigvee : S \mapsto a$ as an operation in $\mathbf{B}$. Dually, $(\bigwedge S = a) \in \mathbf{C}$ implies that $\bigwedge : S \mapsto a$ as an operation in $\mathbf{B}$.

$\mathbf{B}$ is *minimal* if for every other consistent extension $\mathbf{B}_1 = (B, \leq_1, \top, \bot, \bigvee_1, \bigwedge_1)$, we have that $a \leq b$ implies $a \leq_1 b$, $\bigvee S = a$ implies $\bigvee_1 S = a$, and $\bigwedge S = a$ implies $\bigwedge_1 S = a$, for all $a, b \in B$ and all nonempty $S \subseteq B$. We say that $\mathbf{S}$ is *consistent* if it is consistent with some BPPO. In other words, a consistent prespecification is one which can be "extended" to a BPPO.

2.1.3 The consistency decision problem The formal decision problem which we shall address in this subsection is the following.

The decision problem $\mathsf{PreBPPO\text{-}Consis}$:

> **Input:** A BPPO prespecification $\mathbf{S}$.
>
> **Question:** Does $\mathbf{S}$ have a consistent extension to a BPPO?

2.1.4 Acyclicity and the partial order of prespecifications A key issue that we need to address in determining whether or not a prespecification is consistent is to determine whether or not it defines a partial order. If the implied order relation is cyclic, then we cannot have a partial order, which is acyclic by definition. The test is actually accomplished in two steps. In the first step, which we elaborate here, we check for so-called weak acyclicity, in which the underlying graph is checked for cycles.

Let $\mathbf{S} = (P, \mathbf{C})$ be a finite BPPO prespecification. The *graph* of $\mathbf{S}$, denoted $\mathsf{Graph}(\mathbf{S})$, is a directed graph with vertices consisting of the elements of $\mathsf{Aug}(P)$, and edges defined as follows[6].

(i) $(\bot, \top) \in \mathsf{Graph}(\mathbf{S})$.

(ii) For each $a \in P$, $(\bot, a) \in \mathsf{Graph}(\mathbf{S})$ and $(a, \top) \in \mathsf{Graph}(\mathbf{S})$.

(iii) For each constraint of the form $(a < b)$ in $\mathbf{C}$, $(a, b) \in \mathsf{Graph}(\mathbf{S})$.

(iv) For each constraint of the form $(\bigvee S = a)$ in $\mathbf{C}$, and each $s \in S$, $(s, a) \in \mathsf{Graph}(\mathbf{S})$.

(v) For each constraint of the form $(\bigwedge S = a)$ in $\mathbf{C}$, and each $s \in S$, $(a, s) \in \mathsf{Graph}(\mathbf{S})$.

[6]We identify the graph with the set of its edges, so $(x, y) \in \mathsf{Graph}(\mathbf{S})$ means precisely that there is an edge from x to y in the graph.

$\mathbf{S}$ is said to be *weakly acyclic* if $\mathsf{Graph}(\mathbf{S})$ is acyclic. If $\mathbf{S}$ is weakly acyclic, we define the *weak partial order of* $\mathbf{S}$ to be the reflexive and transitive closure $\mathsf{Graph}(\mathbf{S})$, which we denote by $\overline{\mathsf{Graph}}(\mathbf{S})$. In other words, $(a, b) \in \overline{\mathsf{Graph}}(\mathbf{S})$ iff $a = b$ or else there is a directed path from a to b in $\mathsf{Graph}(\mathbf{S})$.

2.1.5 Proposition — complexity of acyclicity issues *Let* $\mathbf{S} = (P, \mathbf{C})$ *be a finite BPPO prespecification.*

(a) *For any consistent extension of* $\mathbf{S}$ *to a BPPO* $\mathbf{B} = (B, \leq, \top, \bot, \bigvee, \bigwedge)$, *we must have that* $\overline{\mathsf{Graph}}(\mathbf{S}) \subseteq\ \leq$. *Thus, if* $\mathbf{S}$ *is consistent, then it must be weakly acyclic.*

(b) *It is decidable in deterministic time* $O(\mathsf{Size}(\mathbf{S}) \cdot \log(\mathsf{Card}(P)))$ *whether or not* $\mathbf{S}$ *is weakly acyclic.*

(c) $\overline{\mathsf{Graph}}(\mathbf{S})$ *may be constructed from* $\mathsf{Graph}(\mathbf{S})$ *in deterministic time* $O(\mathsf{Card}(P)^3)$.

PROOF: Part (a) is immediate. For part (b), the rules of 2.1.4 provide a means of enumerating the tuples (which represent the edges) of $\mathsf{Graph}(\mathbf{S})$. However, a representation of a graph as a set of ordered pairs is not particularly amenable to the efficient detection of cycles. Therefore, it is advantageous to build a more useful representation of the graph. We employ a data structure, with the following characteristics, for representing sets of items.

(i) The entire set of items may be traversed in time linear in the size of the set.

(ii) Given its name, any particular item may be reached in time logarithmic in the size of the set.

(iii) A new item may be added in time logarithmic in the size of the set.

An appropriate data structure is a balanced binary search tree, such as a red-black tree [CLR90, Ch. 14].

This data structure is used in two ways. First of all, the set of all nodes of $\mathsf{Graph}(\mathbf{S})$ is represented in this way. Secondly, for each node a, there is a balanced binary search tree, attached to node a, which represents the set of all nodes which may be reached from a by following a single directed edge from tail to head. Given a way to enumerate the edges of the graph in time linear in the number of tuples (which is easily obtained from the definition of 2.1.4), such a representation may be built in time $O(\mathsf{Size}(\mathbf{S}) \cdot \log(\mathsf{Card}(P)))$. Indeed, the tuples of $\mathsf{Graph}(\mathbf{S})$ themselves may be enumerated in time $O(\mathsf{Size}(\mathbf{S}))$. For each such tuple (a, b), its insertion into the data structure involves two steps. First, a is inserted into the list of all nodes of the graph. Second, b is inserted into the nodes list of a. Each step may be performed in time $O(\log(\mathsf{Card}(P)))$. Thus, to build the graph, $O(\mathsf{Size}(\mathbf{S}))$ steps, each of time complexity $O(\log(\mathsf{Card}(P)))$, must be performed, yielding the complexity $O(\mathsf{Size}(\mathbf{S}) \cdot \log(\mathsf{Card}(P)))$.

Now, to determine whether or not $\mathsf{Graph}(\mathbf{S})$ is acyclic, we start at $\bot$ and traverse the graph depth first. Nodes are marked as they are encountered in the traversal in a forward direction, and the marks are erased upon backtracking over them. The graph is cyclic if we encounter a marked node while traversing in the forward direction; otherwise it is acyclic. In the data structure which we have described, this traversal can be done in time proportional to the number of edges in the graph, which is clearly $O(\mathsf{Size}(\mathbf{S}))$. See, for example [CLR90, Lemma 23.10]. Hence, the whole procedure of building the representation of the graph, and then testing it for acyclicity, is bounded in time complexity by $O(\mathsf{Size}(\mathbf{S}) \cdot \log(\mathsf{Card}(P)))$.

Finally, for part (c), we may employ one of the standard transitive closure algorithms, which run in time $\theta(\mathsf{Card}(P)^3)$ [CLR90, pp. 562-563]. □

2.1.6 Examples — weak acyclicity is not sufficient There may be order dependencies implied by a prespecification $\mathbf{S}$ which are not recaptured in $\overline{\mathsf{Graph}}(\mathbf{S})$. We illustrate with a pair of examples.

Let $\mathbf{S}_1 = (P_1, \mathbf{C}_1)$ be the BPPO prespecification in which $P_1 = \{a, b, c, d, e\}$, and $\mathbf{C}_1 = \{(\bigvee\{a, b, c\} = d), (\bigvee\{a, b\} = e)\}$. Then clearly we must have that $e \leq d$, yet this constraint is not recaptured by $\overline{\mathsf{Graph}}(\mathbf{D})$.

Let $\mathbf{S}_2 = (P_2, \mathbf{C}_2)$ be the BPPO prespecification in which $P_2 = \{a, b, c, d, e\}$, and $\mathbf{C}_2 = \{(\bigvee\{a, b, c\} = d), (\bigvee\{a, b, c\} = e)\}$. This prespecification is inconsistent, because we must have $e = d$. But this inconsistency does not cause $\overline{\mathsf{Graph}}(\mathbf{S})$ to be cyclic, and so is undetected by this test.

Here is a somewhat more complex example. Let $\mathbf{S}_3 = (P_3, \mathbf{C}_3)$ be the BPPO prespecification in which $P_2 = \{a, b, c, d, e, f, g\}$, and $\mathbf{C}_2 = \{(\bigvee\{a, b\} = f), (\bigvee\{b, c\} = f), (\bigvee\{a, c\} = e), (\bigvee\{f, g\} = e)\}$. This prespecification is also inconsistent. Indeed, from $(\bigvee\{a, b\} = f)$ and $(\bigvee\{b, c\} = f)$ we can conclude that $(\bigvee\{a, b, c\} = f)$. Since we have $(\bigvee\{a, c\} = e)$, it must be the case that $e \leq f$. However, $(\bigvee\{f, g\} = e)$ implies that $f \leq e$, whence $e = f$, and so the prespecification is inconsistent. However, $\overline{\mathsf{Graph}}(\mathbf{S})$ is acyclic.

To detect these sorts of relationships between elements, we must perform a more detailed analysis of how the rules interact. This is accomplished by examining the behavior of ideals, as we next describe.

2.1.7 Ideals of prespecifications The notion of an ideal of a BPPO prespecification is very similar to that of an ideal for a BPPO (see 1.3.4), and, in a consistent BPPO prespecification, the two coincide. The details are as follows.

Let $\mathbf{S} = (P, \mathbf{C})$ be a finite BPPO prespecification. A *ideal* of $\mathbf{S}$ is a subset $I \subseteq P$ subject to the following conditions.

(i) $\perp \in I$.

(ii) $x \in I$ and $(y, x) \in \mathsf{Graph}(\mathbf{S})$ implies $y \in I$.

(iii) $S \subseteq I$ and $(\bigvee S = a) \in \mathbf{C}$ implies $a \in I$.

Dually, a *dual ideal* of $\mathbf{S} = (P, \mathbf{C})$ is a subset $I \subseteq P$ subject to the following conditions.

(i) $\top \in I$.

(ii) $x \in I$ and $(x, y) \in \mathsf{Graph}(\mathbf{S})$ implies $y \in I$.

(c) $S \subseteq I$ and $(\bigwedge S = a) \in \mathbf{C}$ implies $a \in I$.

Let $\preceq$ be a partial order on $\mathsf{Aug}(P)$, and let $a \in \mathsf{Aug}(P)$. The *principal preideal* of a with respect to $\preceq$, denoted $\mathsf{PrinIdeal}(a, \preceq)$, is $\{x \in \mathsf{Aug}(P) \mid x \preceq a\}$. Dually, the *dual principal preideal* of a with respect to $\preceq$, denoted $\mathsf{DualPrinIdeal}(a, \preceq)$, is $\{x \in \mathsf{Aug}(P) \mid a \preceq x\}$. We say that $\preceq$ is *stable* with respect to $\mathbf{S}$ if $\overline{\mathsf{Graph}}(\mathbf{S}) \subseteq \preceq$ and every principal preideal I with respect to $\preceq$ is an ideal of $\mathbf{S}$, and every dual principal preideal of $\mathbf{S}$ with respect to $\preceq$ is a dual ideal of $\mathbf{S}$.

Intuitively, stability of $\preceq$ means that applying the rules in $\mathbf{C}$ to principal preideals and dual principal preideals will not add new elements to the sets. In other words, the rules of $\mathbf{C}$ do not imply the necessity of adding new pairs to $\preceq$ to make this partial order consistent with them.

2.1.8 Lemma *Let $\mathbf{S}$ be a finite BPPO prespecification which is weakly acyclic. Then, if $\mathbf{S}$ admits a stable partial order, it admits a smallest such order $\preceq_{\mathbf{S}}$, which is given by $a \preceq_{\mathbf{S}} b$ iff $a \preceq b$ for all stable partial orders $\preceq$ with respect to $\mathbf{S}$.*

PROOF: Assume that $\mathbf{S}$ admits a stable partial order, and let E be the set of all stable partial orders on $\mathsf{Aug}(P)$ with respect to $\mathbf{S}$. Now it is clear that $\cap E$ is a partial order on P, and that it contains $\overline{\mathsf{Graph}}(\mathbf{S})$. We need to show that it is stable. To see this, first of all, note that for each $a \in \mathsf{Aug}(P)$ we have that $\cap_{\preceq \in E}\mathsf{PrinIdeal}(a, \preceq)$ is an ideal of $\mathbf{S}$ which contains a. Indeed, this follows immediately from the easily seen fact that any intersection of ideals is itself an ideal. But $\cap_{\preceq \in E}\mathsf{PrinIdeal}(a, \preceq) = \mathsf{PrinIdeal}(a, \preceq_{\mathbf{S}})$. Dually, we must have for each $a \in \mathsf{Aug}(P)$ that $\cap_{\preceq \in E}\mathsf{DualPrinIdeal}(a, \preceq)$ is a dual ideal of $\mathbf{S}$ which contains a, and that $\cap_{\preceq \in E}\mathsf{DualPrinIdeal}(a, \preceq) = \mathsf{DualPrinIdeal}(a, \preceq_{\mathbf{S}})$. Hence, $(\cap E) = \preceq_{\mathbf{S}}$ is stable, as required. □

2.1.9 Notation If $\mathbf{S}$ is a weakly acyclic finite BPPO prespecification which admits a stable partial order, then we call $\mathbf{S}$ *stable*, and we denote the smallest such stable partial order, as identified in the above lemma, by $\preceq_{\mathbf{S}}$.

It is clear that if $\mathbf{S}$ does not have a stable partial order, then it cannot be extended to a BPPO. Indeed, a stable partial order is the only kind that respects the ordering requirements imposed by the rules. Let us illustrate this more completely with a pair of examples.

2.1.10 Examples Let $\mathbf{S}_1 = (P_1, \mathbf{C}_1)$ be as in 2.1.6. Notice that $e \notin \mathsf{PrinIdeal}(d, \overline{\mathsf{Graph}}(\mathbf{S}_1))$, yet e must be in every ideal which contains a, since every such ideal must also include $\{a, b, c\}$. Hence $\overline{\mathsf{Graph}}(\mathbf{S}_1)$ is not stable with respect to $\mathbf{S}_1$. In this case though, if we add the pair (d, e) to this partial order, we do obtain a stable one.

Now consider $\mathbf{S}_2 = (P_2, \mathbf{C}_2)$ of 2.1.6. In this case, we must have that d is in every ideal containing e, and that e is in every ideal containing d. Since the set of ideals of $\mathbf{S}_2$ must include principal preideals if an associated order $\preceq$ is to be stable, we see that there is no way to extend $\overline{\mathsf{Graph}}(\mathbf{S}_2)$ to a stable partial order. Thus, not every BPPO prespecification with a weakly acyclic partial order may be extended to one with a stable partial order.

Similarly, with $\mathbf{S}_3 = (P_3, \mathbf{C}_3)$ of 2.1.6, we must have that e is in every ideal containing f, and conversely.

2.1.11 Proposition — complexity of deciding stability *Let $\mathbf{S} = (P, \mathbf{C})$ be a finite BPPO prespecification which is weakly acyclic. Then it is decidable in deterministic time $O(\mathsf{Card}(P)^4 \cdot \mathsf{Size}(\mathbf{S}) \cdot \log(\mathsf{Size}(\mathbf{S})))$ whether or not $\mathbf{S}$ has a stable partial order. If it has one, this order may also be computed in deterministic time $O(\mathsf{Card}(P)^4 \cdot \mathsf{Size}(\mathbf{S}) \cdot \log(\mathsf{Size}(\mathbf{S})))$.*

PROOF: The high-level algorithm proceeds as in Figure 2.1.

The idea of the algorithm is as follows. We simply iterate on the principal preideals and dual principal preideals, adding elements which the join and meet rules require to be present, but the partial order has missed. As long as we do not get any cycles, the algorithm succeeds, while a cycle implies that two elements generate identical principal preideals or principal dual preideals, which means that they cannot be distinct elements in any associated BPPO.

It is easy to see that this algorithm is correct. Proceeding a bit more formally, we begin by building the principal ideal and dual principal ideal with respect to $\overline{\mathsf{Graph}}(\mathbf{S})$ for each element $p \in P$. These are represented by $\mathtt{PI}(p)$ and $\mathtt{DPI}(p)$, respectively. Then, we iterate over each rule in $\mathbf{C}$ and each $p \in P$. For a rule of the form $(\bigvee S = a)$, it must be the case that a is in every ideal which contains S. Thus, if each element of S is in $\mathtt{PI}(p)$, but a is not, then we add $\mathtt{PI}(a)$ to $\mathtt{PI}(p)$. Note that we add the entire ideal $\mathtt{PI}(a)$, and not just a. We also add the pair (p, a) to the relation $\preceq$. A similar rule is applied for dual ideals. We repeat this process until an iteration does not produce any modifications. Since the underlying set P is finite, this process must terminate. It is clear that the relation $\preceq$ must be a subset of any stable partial order for $\mathbf{S}$, since it is a minimal one which is closed under application of the rules of $\mathbf{C}$. Thus, once this computation is completed, we just check to see whether the resulting relation is acyclic.

Now let us turn to proving that the complexity is as stated. We assume an ordering on the types, and represent sets as linear linked lists. We start by representing the set S in each rule of the form $(\bigvee S = a)$ and $(\bigwedge S = a)$ as a linearly linked list. This can be done, for all such rules, in time $O(\mathsf{Size}(\mathbf{S}) \cdot \log(\mathsf{Size}(\mathbf{S})))$.

In lines 1-4, for each $p \in P$, we build the principal preideal $\mathsf{PrinIdeal}(p, \overline{\mathsf{Graph}}(\mathbf{S}))$, represented as $\mathtt{PI}(p)$. We can build linear-linked-list representation of the ideal in time $O(\mathsf{Size}(\mathsf{Graph}(\mathbf{S})) \cdot \log(\mathsf{Size}(\mathsf{Graph}(\mathbf{S}))))$. But $\mathsf{Size}(\mathsf{Graph}(\mathbf{S})) \leq \mathsf{Size}(\mathbf{S})$, so this complexity reduces to $O(\mathsf{Size}(\mathbf{S}) \cdot \log(\mathsf{Size}(\mathbf{S})))$. Similarly, we build the representation for each $\mathsf{DualPrinIdeal}(p, \overline{\mathsf{Graph}}(\mathbf{S}))$. This must be done for each element of P, for a total time bounded by $O(\mathsf{Card}(P) \cdot \mathsf{Size}(\mathbf{S}) \cdot \log(\mathsf{Size}(\mathbf{S})))$.

Note that $\mathsf{Size}(\mathbf{S}) \leq \mathsf{Card}(P)^2 + 2 \cdot \mathsf{Card}(P) \cdot 2^{\mathsf{Card}(P)}$, with the $\mathsf{Card}(P)^2$ term representing the contribution of rules of the form $a < b$ and the $2 \cdot \mathsf{Card}(P) \cdot 2^{\mathsf{Card}(P)}$ term representing the size bound on all join and meet rules. But then $\mathsf{Card}(P)^2 +$

```
1.  For each p ∈ P
2.      PI(p) := PrinIdeal(p, Graph‾(S));
3.      DPI(p) := DualPrinIdeal(p, Graph‾(S));
4.  End for each;
5.  ⪯ := Graph‾(S);
6.  done := false;
7.  Repeat
8.      temprel := ∅;
9.      For each p ∈ P
10.         For each (⋁S = a) in C
11.             If S ⊆ PI(p) and a ∉ PI(p)
12.                 then
13.                 temprel := temprel ∪ (a, p);
14.                 PI(p) := PI(p) ∪ PI(a);
15.             End if;
16.         End for each;
17.         For each (⋀S = a) in C
18.             If S ⊆ DPI(p) and a ∉ DPI(p)
19.                 then
20.                 temprel := temprel ∪ (p, a);
21.                 DPI(p) := DPI(p) ∪ DPI(a);
22.             End if;
23.         End for each;
24.     End for each;
25.     If temprel = ∅
26.         then done := true;
27.         else ⪯ := ⪯ ∪ temprel
28.     End if;
29. until done;
30. If ⪯ acyclic then success; else failure; End if.
```

Figure 2.1: Algorithm for 2.1.10

$2 \cdot \mathsf{Card}(P) \cdot 2^{\mathsf{Card}(P)} \leq 4 \cdot \mathsf{Card}(P) \cdot 2^{\mathsf{Card}(P)}$, and so $\mathsf{Size}(\mathbf{S}) \leq 4 \cdot \mathsf{Card}(P) \cdot 2^{\mathsf{Card}(P)}$. Taking the log of both sides, we get that $O(\log(\mathsf{Size}(\mathbf{S}))) \subseteq O(\mathsf{Card}(P))$. Hence the complexity lines 1-4 is bounded by $O(\mathsf{Card}(P)^2 \cdot \mathsf{Size}(\mathbf{S}))$.

Now for the steps in the repeat loop of lines 7-29. A check of a condition of the form $S \subseteq \mathtt{PI}(p)$ may be performed in time $O(\mathsf{Card}(P))$, since both S and $\mathsf{PrinIdeal}(a, \preceq)$ are represented as ordered lists. A similar check applies for the dual preideals. Execution of a union of the form given on line 13 or 20 may be performed in constant time, since we do not do any special structuring at that point. Execution of a union of the form given on line 14 or 21 can be performed in time $O(\mathsf{Card}(P))$, since each ideal and dual ideal is represented as a linearly linked list, and computing the union amounts to building the merger of the associated lists. The number of rules in $\mathbf{C}$ is surely bounded by $\mathsf{Size}(\mathbf{S})$; therefore we may bound the time in each execution of the pair of for loops on lines 10-16 and 17-23 by $O(\mathsf{Card}(P) \cdot \mathsf{Size}(\mathbf{S}))$. The outer for loop spanning lines 9-24 executes $\mathsf{Card}(P)$ times, so the total time for each execution of this outer loop is $O(\mathsf{Card}(P)^2 \cdot \mathsf{Size}(\mathbf{S}))$.

For the if statement on lines 25-28, updating $\preceq$ can take time $O(\mathsf{Card}(P))$, since at most $2 \cdot \mathsf{Card}(P)$ tuples can be included in `temprel` in any one iteration of the outer for loop. This is smaller than the complexity of the outer for loop; thus, the complexity of the body of the repeat loop, spanning lines 7-29, is $O(\mathsf{Card}(P)^2 \cdot \mathsf{Size}(\mathbf{S}))$.

The number of times that the repeat loop is executed is bounded by $\mathsf{Card}(P)^2$, since we must add a tuple to $\preceq$ for the condition `done` to be false, and since $\preceq$ is a relation on P. Thus, the complexity of the entire repeat loop is $O(\mathsf{Card}(P)^4 \cdot \mathsf{Card}(\mathbf{S}))$.

Finally, when the loop is done, we must check to see if the resulting $\preceq$ is acyclic. This may be done in time proportional to the number of pairs in the relation, as explained in the proof of 2.1.5. Thus, the complexity of this step is $O(\mathsf{Card}(P)^2)$. Putting this all together, we get a final complexity of $O(\mathsf{Card}(P)^4 \cdot \mathsf{Size}(\mathbf{S}))$. □

There is one more condition that we need to check. Namely, we need to make sure that the stable partial order that we construct satisfies condition (bppo-iv); that is, that any joins and meets which are defined are lub's and glb's in the partial order. Fortunately, this is guaranteed with a stable partial order. The formalization and proof follow.

2.1.12 Consistency of bounds Let $\mathbf{S} = (P, \mathbf{C})$ be a stable finite BPPO prespecification.

(a) $\mathbf{S}$ is *lub consistent* if, for each rule of the form $(\bigvee S = a) \in \mathbf{C}$, we have that $\mathrm{lub(S)}\!\downarrow$ with $\mathrm{lub(S) = a}$ in $\preceq_{\mathbf{S}}$.

(b) $\mathbf{S}$ is *glb consistent* if, for each rule of the form $(\bigwedge S = a) \in \mathbf{C}$, we have that $\mathrm{glb(S)}\!\downarrow$ with $\mathrm{glb(S) = a}$ in $\preceq_{\mathbf{S}}$.

If $\mathbf{S}$ is both lub and glb consistent, then we say that it is *bounds consistent*.

2.1.13 Proposition — stability implies bounds consistency *Let $\mathbf{S} = (P, \mathbf{C})$ be a stable finite BPPO prespecification. Then, $\mathbf{S}$ is bounds consistent.*

PROOF: This is almost immediate. Let $(\bigvee S = a) \in \mathbf{C}$, and let b be an upper bound for S. Then $S \subseteq \mathsf{PrinIdeal}(b, \preceq_{\mathbf{S}})$. However, since $\preceq_{\mathbf{S}}$ is stable, we must have that $a \in \mathsf{PrinIdeal}(b, \preceq_{\mathbf{S}})$ as well. Thus, $a \preceq b$, and so $\mathbf{S}$ is lub consistent. The property of glb consistency is proved similarly. □

To carry the translation from BPPO prespecification to BPPO to a conclusion, we must add *all* of the join and meet rules implied by those of the prespecification to the final collection, for it is only in this way that we can identify precisely what the generalized join and generalized meet operators will be. The next article identifies this construction.

2.1.14 Extension to full join rules Let $\mathbf{S} = (P, \mathbf{C})$ be a stable finite BPPO prespecification.

(a) The *full join rule set* for $\mathbf{S}$, $\mathsf{FJRS}(\mathbf{S})$, is the smallest set of rules closed under the following operations.

(i) $(\bigvee S = a) \in \mathbf{C}$ implies $(\bigvee S = a) \in \mathsf{FJRS}(\mathbf{S})$.

(ii) If $S \subseteq P$ and $a = \mathrm{lub(S)}$ in $\preceq_{\mathbf{S}}$, then $(\bigvee S = a) \in \mathsf{FJRS}(\mathbf{S})$.

(iii) If $(\bigvee S_1 = a_1), \ldots, (\bigvee S_k = a_k) \in \mathsf{FJRS}(\mathbf{S})$, $(\bigvee\{a_1, \ldots a_k\} = a) \in \mathsf{FJRS}(\mathbf{C})$, then $(\bigvee(\bigcup_{i=1}^{k} S) = a) \in \mathsf{FJRS}(\mathbf{S})$.

(b) The relation $\mathsf{Rel}_{\vee}(\mathbf{S}) \subseteq \mathcal{P}_f(P) \times \mathsf{Aug}(P)$ is defined by $(S, a) \in \mathsf{Rel}_{\vee}(\mathbf{S})$ iff $(\bigvee S = a) \in \mathsf{FJRS}(\mathbf{S})$.

(a′) The *full meet rule set* for $\mathbf{S}$, $\mathsf{FMRS}(\mathbf{S})$, is defined dually as the smallest set of rules closed under the following operations.

(i′) $(\wedge S = a) \in \mathbf{C}$ implies $(\bigwedge S = a) \in \mathsf{FMRS}(\mathbf{S})$.

(ii′) If $S \subseteq P$ and $a = \mathrm{lub(S)}$ in $\preceq_{\mathbf{S}}$, then $(\bigwedge S = a) \in \mathsf{FMRS}(\mathbf{S})$.

(iii′) If $(\bigwedge S_1 = a_1), \ldots, (\bigwedge S_k = a_k) \in \mathsf{FMRS}(\mathbf{S})$, $(\bigwedge\{a_1, \ldots a_k\} = a) \in \mathsf{FMRS}(\mathbf{C})$, then $(\bigwedge \bigcup_{i=1}^{k} S = a) \in \mathsf{FMRS}(\mathbf{S})$.

(b’) The relation $\mathsf{Rel}_{\wedge}(\mathbf{S}) \subseteq \mathcal{P}_f(P) \times \mathsf{Aug}(P)$ is defined by $(S, a) \in \mathsf{Rel}_{\wedge}(\mathbf{S})$ iff $(\bigvee S = a) \in \mathsf{FJRS}(\mathbf{S})$.

Note that the size of these construction may be exponential in the size of the input, but that is not a fatal flaw, since we need not explicitly build these relations. All that we need know is that such an extension exists, and we can determine that by knowing that the prespecification is stable, which can be determined in deterministic polynomial time.

2.1.15 Proposition — defining join and meet in prespecifications *Let $\mathbf{S} = (P, \mathbf{C})$ be a stable finite BPPO prespecification, with $\preceq$ the associated partial order. The following conditions then hold.*

(a) $\mathsf{Rel}_{\vee}(\mathbf{S})$ *and* $\mathsf{Rel}_{\wedge}(\mathbf{S})$ *are partial functions.*

(b) *Any extension of* $\mathbf{S}$ *to a BPPO* $\mathbf{B} = (B, \leq, \top, \bot, \bigvee, \bigwedge)$ *must have* $\mathsf{Rel}_{\vee}(\mathbf{S}) \subseteq \bigvee$ *and* $\mathsf{Rel}_{\wedge}(\mathbf{S}) \subseteq \bigwedge$.

PROOF: This follows immediately from the characterization of $\preceq_{\mathbf{S}}$ in terms of preideals and dual preideals, as given in 2.1.8 and 2.1.9. □

Finally, we are able to formally identify the “natural” BPPO associated with a consistent prespecification.

2.1.16 Consistency and the BPPO of a prespecification Given a stable BPPO prespecification, $\mathbf{S}$, we define the *canonical BPPO of* $\mathbf{S}$ to be $\mathsf{CanBPPO}(\mathbf{S}) = (\mathsf{Aug}(P), \preceq_{\mathbf{S}}, \bot, \top, \mathsf{Rel}_{\vee}(\mathbf{S}), \mathsf{Rel}_{\wedge}(\mathbf{S}))$.

For the terminology to make sense, we must show that $\mathsf{CanBPPO}(\mathbf{S})$ is indeed a BPPO; this and more we do in the following theorem.

2.1.17 Theorem — consistent extensions *Let $\mathbf{S} = (P, \mathbf{C})$ be a finite BPPO prespecification.*

(a) *$\mathbf{S}$ is consistent iff it is stable.*

(b) *If $\mathbf{S}$ is consistent, then the canonical BPPO of $\mathbf{S}$ is the unique minimal consistent extension of $\mathbf{S}$.*

(c) PreBPPO-Consis *may be decided in deterministic polynomial time (in the size of $\mathbf{S}$).*

Proof: We start with (a). If $\mathbf{S}$ is consistent, then in light of the discussion in 1.3.4, the associated BPPO $\mathbf{B}$ must embed into the lattice of all of its ideals. Thus, in particular, the associated partial order must the stable. Conversely, if $\mathbf{S}$ is stable, then we may verify that the canonical BPPO identified in 2.1.16 is indeed a BPPO. Properties (bppo-i) – (bppo-iii) of 1.2.2 are immediate. Property (bppo-iv) follows from bounds consistency (2.1.13). Finally, properties (bppo-vi) and (bppo-vii) follow from the definitions (2.1.14) and consistency (2.1.15) of $\mathsf{Rel}_{\vee}(\mathbf{S})$ and $\mathsf{Rel}_{\wedge}(\mathbf{S})$.

We have argued in the previous paragraph that the canonical BPPO of $\mathbf{S}$ is indeed a BPPO. Unique minimality follows from 2.1.8 and 2.1.15(b).

Part (c) follows from the complexity results of 2.1.5 and 2.1.11. □

Thus, the first critical step in processing a finite BPPO prespecification to determine if it is extendible to a distributive lattice, namely that of determining whether or not it defines a BPPO at all, may be done in deterministic polynomial time.

As we noted above, the canonical BPPO associated with a consistent finite prespecification may be exponentially larger in size than the prespecification (in terms of the number of join and meet rules). Therefore, it is important that we be able to work with the prespecification, without being required explicitly to compute the entire canonical BPPO. Fortunately, this is the case. Indeed, we may even determine when a candidate for a morphism into a lattice is indeed a morphism, just by checking against the prespecification. The following proposition characterizes this explicitly. The proof is a simple consequence of the definitions of the full sets of join and meet rules of 2.1.14, as well as the characterization of stability in 2.1.8. Since we do not explicitly use this result in this report, we omit the simple details of the rather direct proof.

2.1.18 Proposition — characterization of morphisms *Let $\mathbf{S} = (P, \mathbf{C})$ be a consistent finite BPPO prespecification, let $\mathbf{L}$ be a distributive lattice, and let $f : \mathsf{Aug}(P) \to L$ be a function. Then f is a morphism $f : \mathsf{CanBPPO}(\mathbf{P}) \to \mathbf{L}$ iff the following four conditions are satisfied*

(i) *For each* $p \in \mathsf{Aug}(P)$, $f(\bot) \leq f(p)$ *and* $f(p) \leq f(\top)$.

(ii) *For each constraint* $(a < b) \in \mathbf{C}$, $f(a) \leq f(b)$.

(iii) *For each constraint* $(\bigvee S = a) \in \mathbf{C}$, $f(a) = \bigvee f(S)$.

(iv) *For each constraint* $(\bigwedge S = a) \in \mathbf{C}$, $f(a) = \bigwedge f(S)$. □

2.1.19 Terminology of BPPO's extended to consistent prespecifications As a terminological convenience, we extend the use of names of properties of BPPO's to consistent BPPO prespecifications. The meaning is that the consistent prespecification $\mathbf{S}$ has the property iff $\mathsf{CanBPPO}(\mathbf{S})$ does. Thus, for example, if we say that $\mathbf{S}$ is separable, we mean that $\mathsf{CanBPPO}(\mathbf{S})$ is separable.

2.2 The Basic NP-Complete Extension Problems

Being able to extend a finite BPPO prespecification to a BPPO means that the prespecification may be extended to a lattice. This follows directly from 1.3.5. From the main results of the previous subsection, we know that deciding whether or not this is possible can be done in deterministic polynomial time. However, if we demand that that extension be distributive, things become much more difficult. Indeed, in this subsection, we shall show that demanding that the extension be distributive causes the extension problem to become NP-complete. This is irrespective of the kind of distributive extension we seek, as long as it is nontrivial. We begin with a formalization of the problem statement.

2.2.1 Definitions of the basic problems We consider two basic decision problems, Dist-BinExt and Dist-InjExt. In each case, a problem instance is a BPPO prespecification. Because of their importance, we highlight these problems clearly below.

The decision problem Dist-BinExt:

Input: A BPPO prespecification $\mathbf{S}$.

Question: Does $\mathbf{S}$ have a consistent extension to a BPPO, which in turn has an extension to any of the distributive extension categories?

The decision problem Dist-InjExt:

Input: A BPPO prespecification $\mathbf{S}$.

Question: Does $\mathbf{S}$ have an extension to a consistent BPPO, which in turn has an injective extension to any of the distributive extension categories?

In light of 1.4.10, the answer to a query in each of these three problem categories is independent of which distributive extension category we speak. And, by 1.4.12, each question above is also equivalent to asking whether or not the prespecification has a *universal* extension of the given type to the given category. In other words, the three problems listed above cover all of the questions about extensions to distributive extension categories which we consider (except for questions involving constant types).

We work with BPPO prespecifications rather than with BPPO's themselves, because the former represent the form in which such structures are typically specified in computational applications, such as feature structure languages like CUF. The actual BPPO corresponding to a consistent prespecification may be exponentially larger in size, because of the tremendous expansion that the generalized associativity rules can effect. Thus, to work with BPPO's themselves might yield an artificially low complexity measure.

It is useful to know the complexity of the problems when restricted to certain subsets of BPPO prespecifications. There are three particularly useful measures in this regard. Given a BPPO prespecification $\mathbf{S} = (P, \mathbf{C})$, the *height* of $\mathbf{S}$ is the length of the longest directed path in $\overline{\mathsf{Graph}}(\mathbf{S})$. Note that this may be strictly less than the length of the longest path in $\preceq_{\mathbf{S}}$, as may be seen by considering the example $\mathbf{S}_1$ of 2.1.6 and 2.1.10. Next, given a BPPO prespecification, the $\bigvee$*-fanout* (or *join fanout*) is the maximum of 1 and cardinality of S in the largest rule of the form $(\bigvee S = a) \in \mathbf{R}$. Dually, the $\bigwedge$*-fanout* (or *meet fanout*) is the maximum of 1 and cardinality of S in the largest rule of the form $(\bigwedge S = a) \in \mathbf{R}$. Again, the fanout of the BPPO of a consistent prespecification can be much larger than that of the BPPO itself.

We formulate problems whose inputs are restricted to prespecifications bounded by height and fanout. To highlight these problems, we list them below.

The decision problem $\mathsf{Dist\text{-}BinExt}(h, j, m)$:

Input: A BPPO prespecification $\mathbf{S}$, with height bounded by h, $\bigvee$-fanout bounded by j, and $\bigwedge$-fanout bounded by m.

Question: Does $\mathbf{S}$ have an consistent extension to a BPPO, which in turn has an extension to any of the distributive extension categories?

The decision problem $\mathsf{Dist\text{-}InjExt}(h, j, m)$:

Input: A BPPO prespecification $\mathbf{S}$, with height bounded by h, $\bigvee$-fanout bounded by j, and $\bigwedge$-fanout bounded by m.

Question: Does $\mathbf{S}$ have an extension to a consistent BPPO, which in turn has an injective extension to any of the distributive extension categories?

To allow for the case in which we do not place restrictions on some of these parameters, we also allow the argument ∞. Thus, $\mathsf{Dist\text{-}BinExt}(\infty, 3, 3)$, is the problem $\mathsf{Dist\text{-}BinExt}$ restricted to prespecifications in which the $\bigvee$- and $\bigwedge$-fanout are each restricted to be no larger than 3, but with no restriction on the height. Of course, $\mathsf{Dist\text{-}BinExt}(\infty, \infty, \infty)$ is just $\mathsf{Dist\text{-}BinExt}$.

We begin with the "easy" part of the proof of NP-completeness — namely, that the problems under consideration are in NP.

2.2.2 Proposition *The problems* Dist-BinExt *and* Dist-InjExt *are each in NP.*

PROOF: Let $\mathbf{S}$ be a BPPO prespecification. First of all, by 2.1.16(c), we know that we can determine in deterministic polynomial time whether or not $\mathbf{S}$ is consistent. Now, if is consistent, we can simply guess at a solution and test. We use the characterization of distributive extensions in terms of ideal binary decompositions. By 1.4.7, we know that $\mathsf{CanBPPO}(\mathbf{S})$ has an extension pair iff it has an ideal binary decomposition. But to test, we simply guess at such a decomposition, which, in view of the characterization of 1.4.2, amounts to selecting a partition of $\mathsf{Aug}(P)$ and seeing if it defines a morphism into $\mathbf{2}$. But such a test, in view of 2.1.17, may be performed in time which is deterministic polynomial in the size of $\mathbf{S}$. Hence Dist-BinExt is in NP.

The proof that Dist-InjExt is in NP is similar. This time, we use the characterization of 1.4.10 which characterizes injective decompositions in terms of separating sets of ideal binary decompositions. But it is easy to see that such a separating set need have at most $(n^2 - n)/2$ members, where n is the cardinality of $\mathsf{Aug}(P)$. Indeed, there are $(n \cdot (n-1))/2$ pairs of elements in $\mathsf{Aug}(P)$, and we need but one morphism to separate each. So, we just guess at such a separating set, and test each one, as described above, to see if it both separates the appropriate pair and is a morphism. Hence, Dist-InjExt is also in NP. □

Observe that the above result also implies that all problems of the form $\mathsf{Dist\text{-}BinExt}(h, j, m)$, $\mathsf{Dist\text{-}SpecExt}(h, j, m)$ and $\mathsf{Dist\text{-}InjExt}(h, j, m)$ are in NP, for $1 \leq h, j, m \leq \infty$, since they are restrictions of the problems studied in the above proposition.

2.2.3 Notation for propositional formulas We assume a general familiarity with propositional logic, particularly with the satisfiability problems associated with modern computational complexity theory, as may be found in [GJ79] or [AHU74]. Here we just clarify notation and terminology. Given a propositional formula φ, $\mathrm{Vars}(\varphi)$ denotes the set of (Boolean) variables (or proposition letters) which explicitly occur in φ. Given a set X of variables, we may say that φ is *taken* over X provided $\mathrm{Vars}(\varphi) \subseteq X$. The possibility that $\mathrm{Vars}(\varphi)$ is a proper subset of X is not excluded. A *literal* is either a variable x or else its negation $\neg x$, and $\mathrm{Var}(\ell)$ denotes the variable of the literal ℓ. The *literal set of* X, denoted $\mathsf{Literals}(X)$, is $X \cup \{\neg x \mid x \in X\}$. The *complement* of a literal is its negation; thus, the complement of x is $\neg x$, and the complement of $\neg x$ is x. A *clause* is a disjunction $(\ell_1 \vee \ldots \vee \ell_k)$ of literals. A clause is *trivial* if it contains both a variable and its negation as literals. (A trivial clause is always true.) A *unit clause* is a clause which is the disjunction of exactly one literal. A formula is in *conjunctive normal form* (or *CNF*) if it is a conjunction of clauses; a CNF formula is *nonredundant* if it does not contain any trivial clauses, clauses with duplicate literals, duplicate clauses, or pairs of distinct clauses for which the literals of one are a subset of the literals of the other. Note that we can always render a formula nonredundant in linear time. Given a natural number k, a formula is in k-CNF if it is in CNF, and no clause contains more than k literals. It is in *strict k-CNF* if each clause contains exactly k literals. A CNF formula is *unit free* if it does not contain any unit clauses. A formula is in *Negation Normal Form* (or *NNF*), if it is built up from the connectives $\vee$, $\wedge$, and $\neg$, with negation occurring only at the level of atoms. (In other words, it is built up from literals using only the connectives $\vee$ and $\wedge$.) The *length* of a formula is just the length of the string representing it.

Let X be a set of variables. A *truth assignment* for X is a function $f : X \to \{\bot, \top\}$.[7] If φ is a formula with $\mathrm{Vars}(\varphi) \subseteq X$, then we also call f an *interpretation* (with respect to X) for φ. Given a truth assignment f for X and a formula φ with $\mathrm{Vars}(\varphi) \subseteq X$, we define the *truth value* $\bar{f}(\varphi) \in \{\bot, \top\}$ of φ for f in the usual fashion. The interpretation f is called a *model* of φ if $\bar{f}(\varphi) = \top$, and φ is *satisfiable* if it has a model. $\mathsf{Mod}(\varphi)$ denotes the set of all models of φ. Two formulas φ_1 and φ_2 are *logically equivalent* with respect to X if $\mathsf{Mod}(\varphi_1) = \mathsf{Mod}(\varphi_2)$.

The *CNF satisfiability problem* Sat-CNF takes as input a formula in CNF and answers true if it is satisfiable, and false otherwise. The *nonredundant CNF satisfiability problem* Sat-Nonredundant-CNF is defined similarly, but for nonredundant CNF formulas only. The *k-CNF satisfiability* problem, Sat-kCNF, (resp. *exact k-CNF satisfiability* problem Sat-Exact-kCNF) is defined similarly, but for k-CNF (resp. exact k-CNF) clauses. It is well-known that Sat-CNF and Sat-Exact-3CNF are each NP-complete problems [GJ79, Thms. 2.6, 3.1], [AHU74, Thms. 10.3, 10.4]. It follows trivially that Sat-kCNF and Sat-Exact-kCNF are NP-complete for any $k > 3$, as is Sat-Nonredundant-CNF. The *2-CNF satisfiability* problem, Sat-2CNF, is defined similarly, but is known to be solvable in deterministic polynomial time [EAS76, Sec. 2], from which it trivially follows that Sat-Exact-2CNF is also solvable in deterministic polynomial time.

In 1.4.9, we introduced the notion of separability for a BPPO, and used it to characterize the existence of distributive extensions. Since our approach to showing that the distributive extension problems are NP-hard is to transform satisfiability problems to separability problems, we begin by introducing a corresponding notion of separability for propositional formulas, which we will use in the transformation.

2.2.4 Separability and strong separability Let φ be a formula over the set $X = \{x_1, \ldots, x_n\}$ of variables.

(a) A *separator* for a pair $\{x_i, x_j\} \subseteq X$ is a two-element set $\{f_=, f_{\neq}\} \subseteq \mathsf{Mod}(\varphi)$ with the property that

$$f_=(x_i) = f_=(x_j);$$
$$f_{\neq}(x_i) \neq f_{\neq}(x_j).$$

A *separator* for $\{x_i, \bot\}$, with $x_i \in X$, is a singleton $\{f\} \subseteq \mathsf{Mod}(\varphi)$ with the property that $f(x_i) = \top$. Dually, a *separator* for $\{x_i, \top\}$, with $x_i \in X$, is a

[7] We use $\bot$ to represent **false** and $\top$ to represent **true**. This convention will make translations between logic and and BPPO prespecifications smoother, and the meaning of the symbols $\top$ and $\bot$ will always be clear from context. Also, when $z \in \{\bot, \top\}$, with z representing a truth value, we write $\neg z$ to denote $\bot$ if $z = \top$ and to denote $\top$ if $z = \bot$.

singleton $\{f\} \subseteq \mathsf{Mod}(\varphi)$ with the property that $f(x_i) = \bot$. Any singleton $\{f\} \subseteq \mathsf{Mod}(\varphi)$ is a separator for $\{\bot, \top\}$. Given a set Q of pairs of elements from $X \cup \{\bot, \top\}$, a *Q-separator* for φ is a set $\Delta \subseteq \mathsf{Mod}(\varphi)$ which includes as a subset a separator for each pair in Q. A *(full) separator* for φ (with respect to X) is a set $\Delta \subseteq \mathsf{Mod}(\varphi)$ which includes as a subset a separator for every pair of distinct elements in $X \cup \{\bot, \top\}$. (Here $\bot$ and $\top$ are just special symbols; think of them as variables whose values are fixed. Their inclusion is a technicality which will be essential in translation of these concepts to BPPO's.)

(b) A *strong separator* for a pair $\{x_i, x_j\} \subseteq X$ is a four-element set $\{f_{\bot\bot}, f_{\bot\top}, f_{\top\bot}, f_{\top\top}\} \subseteq \mathsf{Mod}(\varphi)$ such that

$$\begin{array}{ll} f_{\bot\bot}(x_i) = \bot; & f_{\bot\bot}(x_j) = \bot; \\ f_{\bot\top}(x_i) = \bot; & f_{\bot\top}(x_j) = \top; \\ f_{\top\bot}(x_i) = \top; & f_{\top\bot}(x_j) = \bot; \\ f_{\top\top}(x_i) = \top; & f_{\top\top}(x_j) = \top. \end{array}$$

A *(full) strong separator* for φ (with respect to X) is a set $\Delta \subseteq \mathsf{Mod}(\varphi)$ which includes as a subset a strong separator for every pair of distinct variables in X, and, for each $x_i \in X$, Δ includes as elements a separator for $\{x_i, \bot\}$ and a separator for $\{x_i, \top\}$. Under this definition, every strong separator is a separator. Note that if φ is consistent and X has at least two distinct elements, then the condition that Δ include elements which separate $\{x_i, \top\}$ and $\{x_i, \bot\}$ for each $x_i \in X$ is redundant, since for every variable x_i, there must be models $f, g \in \mathsf{Mod}(\varphi)$ with $f(x_i) = \bot$ and $g(x_i) = \top$. This latter property will often be used in proofs in which the nontrivial cases involve at least two variables.

(c) Given $S \subseteq X$, a set $\Delta \subseteq \mathsf{Mod}(\varphi)$ *renders S atomic* if for each $a \in S$, there is exactly one $f \in \Delta$ with the property that $f(a) = \top$. (This concept will be used in Section 2.5 in connection with prespecifications involving atoms. Notice the close connection with the definition of the same name for decompositions of BPPO's in 1.5.3.) Now if Q is a set of pairs of elements of $X \cup \{\bot, \top\}$, then a set $\Delta \subseteq \mathsf{Mod}(\varphi)$ is a (Q, S)-separator for φ if it is a Q-separator which also renders S atomic.

Based upon these definitions, we introduce four new decision problems. (Here k denotes any natural number larger than 1.)

The decision problems $\mathsf{Sep\text{-}CNF}$ (resp. $\mathsf{Sep\text{-}kCNF}$):

Input: A propositional formula φ, in CNF (resp. k-CNF) over a set X of variables.

Question: Is φ separable with respect to X?

The decision problem $\mathsf{StrongSep\text{-}CNF}$ (resp. $\mathsf{StrongSep\text{-}kCNF}$):

Input: A propositional formula φ, in CNF (resp. k-CNF) over a set X of variables.

Question: Is φ strongly separable with respect to X?

One of the keys to our reduction proofs is to transform the property of satisfiability to that of strong separability. The following formula transformations are central to this.

2.2.5 Special formulas Let φ be any NNF formula over the set $X = \{x_1, \ldots, x_n\}$ of variables. Let $Y = \{y_1, \ldots, y_n\}$ be another set of variables of the same size, with $X \cap Y = \emptyset$.

(a) Define $\varphi^{\pi_{01}}$ to be the formula obtained from φ by replacing each literal by its complement.

(b) Define $\varphi^{\pi_{XY}}$ to be the formula obtained from φ obtained by replacing each occurrence of x_i with y_i, for $1 \le i \le n$. (Note specifically that all members of X need *not* occur in φ for this to make sense.)

(c) Define $\tilde{\varphi} = (\varphi \vee \varphi^{\pi_{XY}}) \vee (\varphi \vee \varphi^{\pi_{XY}})^{\pi_{01}}$.

2.2.6 Lemma *Let φ be any NNF formula over the set $X = \{x_1, \ldots, x_n\}$ of variables. Then the following conditions are equivalent.*

(a) *$\tilde{\varphi}$ is strongly separable.*

(b) *$\tilde{\varphi}$ is separable.*

(c) *φ is satisfiable.*

PROOF: ((a) $\Rightarrow$ (b)) This follows from the discussion of 2.2.4.
((b) $\Rightarrow$ (c)) $\tilde{\varphi}$ is satisfiable iff φ is. Thus, if $\tilde{\varphi}$ is separable, φ must be satisfiable.
((c) $\Rightarrow$ (a)) Suppose that φ is satisfiable. If φ contains fewer than two distinct variables, then the result is immediate. If φ contains at least two distinct variables, then it suffices to show that any two variables in $X \cup Y$ are strongly separable. Let $f \in \mathsf{Mod}(\varphi)$. Then f is totally independent of the variables in Y (since the variables in Y do not occur in φ), and so, regarding f as an interpretation for $\tilde{\varphi}$, we have that any two variables in Y are strongly separable, since we can choose the truth values of the variables in Y as we please without risking the loss of the property of being a model of φ. Similarly, if we pick any $f \in \mathsf{Mod}(\varphi^{\pi_{XY}})$, and regard it as a model of $\tilde{\varphi}$, then we see that any two elements of X are strongly separable. Now, let $x \in X$ and $y \in Y$. Pick any $f \in \mathsf{Mod}(\varphi)$. (Thus, f depends only upon variables in X, and is independent of variables in Y.) Define $g : X \cup Y \to \{\bot, \top\}$ by

$$g(v) = \begin{cases} f(v) & \text{if } v \in X; \\ \neg f(v) & \text{if } v \in Y. \end{cases}$$

Then $g \in \mathsf{Mod}(\varphi)$. Now define $h : X \cup Y \to \{\bot, \top\}$ by

$$h(v) = \begin{cases} \neg f(v) & \text{if } v \in X; \\ f(v) & \text{if } v \in Y. \end{cases}$$

and $k : X \cup Y \to \{\bot, \top\}$ by $k(v) = \neg f(v)$. Then $h, k \in \mathsf{Mod}(\varphi^{\pi_{01}}) \subseteq \mathsf{Mod}((\varphi \vee \varphi^{\pi_{XY}})^{\pi_{01}})$. Furthermore, $\{f, g, h, k\}$ is a strong separator for $\{x, y\}$, as required. □

2.2.7 Proposition *Let φ be a CNF formula. Then $\tilde{\varphi}$ may be translated into CNF in deterministic time polynomial in the size of φ.*

PROOF: Since $\tilde{\varphi}$ has only four disjuncts, the standard transformation of distributing the disjunction over the conjunction yields a formula which is $O(s^4)$ in size, with s denoting the size of the original formula. □

We are now ready to prove that Sep-CNF and StrongSep-CNF are NP-complete. While these are not our ultimate results, the NP-completeness of these intermediate results is critical to showing the NP-completeness of the distributive extension problems.

2.2.8 Theorem — NP-completeness of separability testing *The problems* Sep-CNF *and* StrongSep-CNF *are NP-complete.*

PROOF: First of all, let us show that these problems are in NP. Let φ be a formula over the set $X = \{x_1, \ldots, x_n\}$ of variables. If $n < 2$, then the problem is trivial. So, assume that X contains at least two distinct variables. Then a separator for φ need contain at most $n^2 - n$ models. Indeed, there are $(n \cdot (n-1))/2$ ways to pick a pair of variables from X, and we need at most two model for each. So, we pick any set $\Delta = \{f_{ij} \mid 1 \le i < j \le n\} \cup \{g_{ij} \mid 1 \le i < j \le n\}$ of interpretations over X, and we test to see whether $\{f_{ij}, g_{ij}\}$ separates $\{x_i, x_j\}$. We can observe immediately whether or not $f_{ij}(x_i) \neq f_{ij}(x_j)$ and whether $g_{ij}(x_i) = g_{ij}(x_j)$, and we can test whether or not each is a model in time linear in the size of φ, just by substituting and evaluating. Thus, we can test all $n^2 - n$ interpretations in deterministic time $O(\mathsf{Size}(\varphi) \cdot n^2)$. For strong separability, the process is essentially the same, except that we need to test $2 \cdot (n^2 - n)$ interpretations. Hence both problems are in NP.

The NP hardness of StrongSep-CNF is a consequence of 2.2.6 and 2.2.7, since we know that Sat-CNF is NP complete, and these two results combined show that we can transform Sat-CNF to StrongSep-CNF in polynomial time. The NP hardness of Sep-CNF follows from the fact that $\tilde{\varphi}$ is separable iff it is satisfiable (2.2.6).

Since these problems are both in NP and are NP-hard, they are NP-complete. □

We now turn to the problem of reducing separability of logical formulas to separability of BPPO's. We begin by associating a BPPO prespecification with a CNF formula.

2.2.9 The BPPO prespecification of a unit-free CNF formula Let $X = \{x_1, \ldots, x_n\}$ be a finite set of variables, and let $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_k$ be a unit-free CNF formula over X. The *BPPO prespecification* of φ, denoted $\mathbf{S}_\varphi = (P_\varphi, \mathbf{C}_\varphi)$, is defined as follows.

(i) $P_\varphi = \mathsf{Literals}(X)$.

(ii) $\mathbf{C}_\varphi$ consists of the following rules.

- For each $x \in X$, $(\bigvee\{x, \neg x\} = \top), (\bigwedge\{x, \neg x\} = \bot) \in \mathbf{C}_\varphi$.
- For each clause $\varphi_i = (\ell_{i1} \vee \ldots \vee \ell_{i(m_i)})$, $(\bigvee\{\ell_{i1}, \ldots, \ell_{i(m_i)}\} = \top) \in \mathbf{C}_\varphi$.

Note that literals, regarded as members of P_φ, are formal entities, and have no special logical significance as members of the set of inner types. Also, not the the property of unit freeness is essential. Otherwise, the definition of the rules in $\mathbf{C}_\varphi$ would not be well-defined, as the set S in a rule of the form $(\bigvee S = a)$ or $(\bigwedge S = a)$ must contain at least two distinct elements. However, this restriction will not prove to be a problem.

2.2.10 Example It is not easy to give exact graphical representations of BPPO's or their prespecifications, because lub's and glb's are not always joins and meets, respectively. But, with a little care, we can illustrate the general idea, and it is helpful to envision how the above construction works. Figure 2.2 presents the rough idea for the formula $(a \vee b \vee c) \wedge (\neg a \vee b \vee \neg c)$. Think of the nodes labelled $(\top)$ as being the same as the top node $\top$, and the nodes labelled $(\bot)$ as being the same as the bottom node $\bot$. We have drawn them separately to highlight the joins and meets which are present. Lines without arrowheads link nodes which are actually the same, while lines with arrowheads link distinct nodes. Note that we have a join to $\top$ and a meet to $\bot$ for each pair $\{a, \neg a\}$, $\{b, \neg b\}$, and $\{c, \neg c\}$. We also have a join to $\top$ for the literals corresponding to each of the two clauses: $\{a, b, \neg c\}$ and $\{\neg a, b, c\}$. In other words, in addition to the obvious rules relating each variable to its complement, we have a join rule of the form $(\bigvee S = \top)$ for each literal whose set of literals is S. Note that the height of this prespecification is only 2, and its fanout is the maximum of 2 and the number of literals in the largest clause, which is 3 in the case of this example.

It is clear from this construction that the resulting BPPO prespecification will be weakly acyclic; we record this as a formal fact.

2.2.11 Lemma *Let φ be a unit-free CNF propositional formula. Then $\mathbf{S}_\varphi$ is weakly acyclic.* □

What is more interesting is that the BPPO prespecification is almost the BPPO, in the sense that only trivial new rules are added, and $\overline{\mathsf{Graph}}(\mathbf{S}_\varphi)$ is the partial order of the BPPO. Thus, the representation is a particularly simple one in this case.

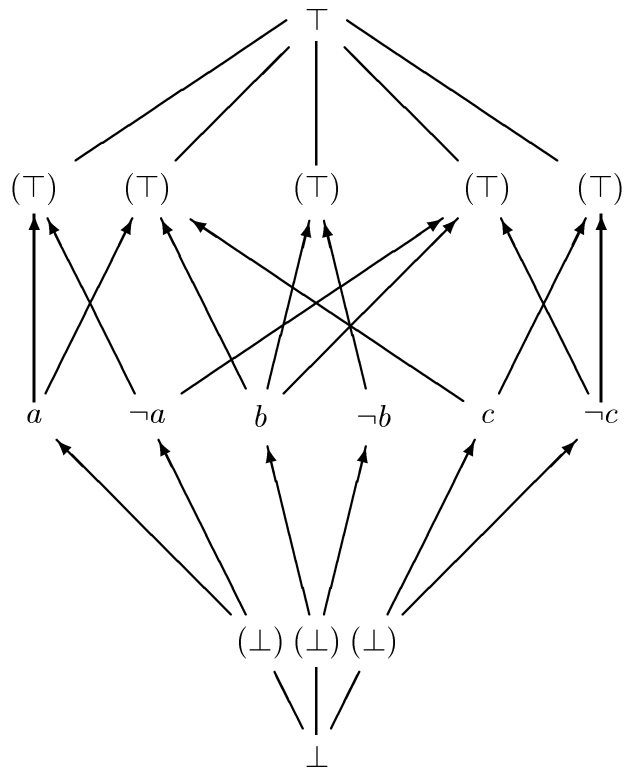

Figure 2.2: The BBPO prespecification associated with $(a \vee b \vee c) \wedge (\neg a \vee b \vee \neg c)$.

2.2.12 Lemma *Let φ be a unit-free CNF propositional formula. Then, if φ is satisfiable, $\mathbf{S}_\varphi$ is consistent, and we furthermore have the following.*

(a) *The height of $\mathbf{S}_\varphi$ is 2.*

(b) *A rule R is in $\mathsf{Rel}_\vee(\mathbf{S}_\varphi)$ iff there are rules $(\bigvee S_1 = \top), \ldots (\bigvee S_k = \top) \in \mathbf{C}_\varphi$ such that $R = (\bigvee(\bigcup_{i=1}^k S_i) = \top)$. Similarly, a rule is in $\mathsf{Rel}_\wedge(\mathbf{S}_\varphi)$ iff there are rules $(\bigwedge S_1 = \bot), \ldots (\bigwedge S_k = \bot) \in \mathbf{C}_\varphi$ such that $R = (\bigwedge(\bigcup_{i=1}^k S_i) = \bot)$.*

(c) *For $a, b \in \mathbf{S}_\varphi$, $a \preceq_{\mathbf{S}_\varphi} b$ iff $(a, b) \in \overline{\mathsf{Graph}}(\mathbf{S}_\varphi)$.*

PROOF: Claim (a) is immediate, because every join rule is of the form $(\bigvee S = \top)$, and every meet rule is of the form $(\bigwedge S = \bot)$. From claim (a), it follows that the only way that a generalized associativity law could be used to generate a new join rule would be to have two rules of the form $(\bigvee S_1 = \top)$ and $(\bigvee S_2 = \top)$ combine to yield $(\bigvee(S_1 \cup S_2) = \top)$. New meet rules may be generated only by the dual construction. Thus, claim (b) is satisfied. Claim (c) is now immediate as well, since the only way that $\overline{\mathsf{Graph}}(\mathbf{S}_\varphi)$ can fail to be $\preceq_{\mathbf{S}_\varphi}$ is to have the join and meet rules contribute new entries, as shown in the proof of 2.1.11. □

The next lemma is crucial, because it not only provides a strong connection between satisfiability of a logical formula and consistency of the corresponding BPPO prespecification, but it also shows that the models of the formula correspond to the ideal binary decompositions of the prespecification. This provides the key that we need to show that the distributive extension problems are NP-complete.

2.2.13 Lemma *Let φ be a unit-free CNF propositional formula. Then φ is satisfiable iff $\mathbf{S}_\varphi$ is consistent and $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$ is somewhere separable. Furthermore, whenever φ is consistent, there is a natural correspondence between $\mathsf{Mod}(\varphi)$ and the ideal binary decompositions of $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$, given by sending each $f \in \mathsf{Mod}(\varphi)$ to the ideal binary decomposition (I, J) with*

(a) $I = \{p \in \mathsf{Aug}(P_\varphi) \mid \bar{f}(p) = \bot\}$;

(b) $J = \{p \in \mathsf{Aug}(P_\varphi) \mid \bar{f}(p) = \top\}$.

PROOF: First of all, assume that φ is satisfiable. Then, in view of part (c) of the previous lemma, we can conclude that every principal preideal of $\mathbf{S}_\varphi$ is in fact an ideal. Hence, by definition, $\preceq_{\mathbf{S}_\varphi}$ is stable, and hence $\mathbf{S}_\varphi$ is consistent, by 2.1.17(a). Next, let $f \in \mathsf{Mod}(\varphi)$. Define the function $g : P_\varphi \to \{\bot, \top\}$ to be the restriction of $\bar{f}$ to P_φ. Then, given the characterization of join and meet rules in part (b) of the previous lemma, we see that g must be the underlying function of a morphism $g : \mathsf{CanBPPO}(\mathbf{S}_\varphi) \to \mathbf{2}$. Hence, by 1.4.2, $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$ has an ideal binary decomposition, and so is somewhere separable.

Conversely, assume that $\mathbf{S}_\varphi$ is consistent and $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$ has an ideal binary decomposition (I, J). Define $f : X \to \{\bot, \top\}$ to be the restriction of $\mathsf{Ifn}_{(I,J)}$ to X. It is immediate from the structure of $\mathbf{S}_\varphi$ that $f \in \mathsf{Mod}(\varphi)$. Hence, by 2.2.12(c), $f \in \mathsf{Mod}(\varphi)$, and so φ is satisfiable.

The correspondence between $\mathsf{Mod}(\varphi)$ and the ideal binary decompositions of $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$ follows from this construction. □

We are finally in position to prove our main results. We can actually prove that the problem Dist-BinExt is NP-complete even for join fanout 3 at this point, because only satisfiability and not separability is involved. On the other hand, we cannot put a bound on join fanout for Dist-InjExt at this point, because our transformation from a satisfiable CNF formula to a separable one does not preserve the property of having three literals per clause.

2.2.14 Theorem — NP-completeness of Dist-BinExt *The problem* Dist-BinExt$(2, 3, 2)$ *is NP-complete. Thus, in particular,* Dist-BinExt *is NP-complete.*

PROOF: We have already shown in 2.2.2 that the problems are in NP. To complete the proof, let φ be an exact 3-CNF propositional formula. Then φ is trivially unit free. Now, from 2.2.13, we know that we can transform the satisfiability problem for φ into the question of whether $\mathbf{S}_\varphi$ is consistent and $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$ has an ideal binary decomposition. However, we know that we can perform the test of whether or not $\mathbf{S}_\varphi$ is consistent in deterministic polynomial time (2.1.17). Thus, since Sat-Exact-3CNF is NP-complete, it must also be the case that deciding whether $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$ has an ideal binary decomposition is NP-complete. But this latter question is equivalent to the one of asking whether $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$ has a distributive extension, by 1.4.7.

Furthermore, we have by 2.2.12 that the height of $\mathbf{S}_\varphi$ is 2. The join fanout of 3 and meet fanout of two follow directly from the fact that φ is in 3-CNF. Thus, we have that $\mathsf{Dist\text{-}BinExt}(2, 3, 2)$ is NP-complete. Since $\mathsf{Dist\text{-}BinExt}$ is more general, but also in NP, it must be NP-complete as well. □

The final key to establishing the NP-hardness of the injective distributive extension problem is to show that separability in logical formulas corresponds to separability of BPPO's.

2.2.15 Proposition — separability is separability *Let φ be a unit-free CNF propositional formula. Then φ is separable iff $\mathbf{S}_\varphi$ is consistent and its minimal extension $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$ to a BPPO has a separating set of ideal binary decompositions.*

PROOF: First of all, assume that φ is separable. Then, in particular, it is satisfiable, and so by 2.2.13 we know that $\mathbf{S}_\varphi$ is consistent and has an ideal binary decomposition. But, consider the structure of $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$. The only elements, other than $\bot$ and $\top$, are the elements corresponding to the variables $X = \{x_1, \ldots, x_n\}$ over which φ is defined, and the set $\{\neg x_1, \ldots, \neg x_n\}$ of its complements. Now, by 2.2.13, we have a natural correspondence between elements of $\mathsf{Mod}(\varphi)$ and ideal binary decompositions. For a pair of the form $\{x, \neg x\}$, any ideal binary decomposition will separate them, since they are complements of each other. For pairs of literals $\{\ell_1, \ell_2\}$ arising from distinct variables, the fact that φ is separable ensures that there is an $f \in \mathsf{Mod}(\varphi)$ with $f(\ell_1) \neq f(\ell_2)$. But, under the correspondence described in 2.2.13, the images of these elements must lie in different components of the ideal binary decomposition defined by f. Thus, x and y are separable in $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$. Similarly, the pair $\{\neg x, \neg y\}$ is separable. Finally, the condition of bivaluedness within separability ensures that $\bot$ and x, and $\top$ and x, will be separable in $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$. Hence, $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$ has a separating set of ideal binary decompositions.

Conversely, assume that $\mathbf{S}_\varphi$ is consistent and that $\mathsf{CanBPPO}(\mathbf{S}_\varphi)$ has a separating set of ideal binary decompositions. Then, for each such ideal binary decomposition (I, J), if we define a function $f : X \to \{\bot, \top\}$ by

$$f(x) = \begin{cases} \top & \text{if } x \in I; \\ \bot & \text{if } x \in J, \end{cases}$$

it is easy to see that f is a model of φ. But since the set of ideal binary decompositions is separating, so too will be the corresponding set of models of φ. Thus, φ is separable, as required. □

2.2.16 Theorem — NP-completeness of $\mathsf{Dist\text{-}InjExt}$ *The problem $\mathsf{Dist\text{-}InjExt}(2, \infty, 2)$ is NP-complete. Thus, in particular, $\mathsf{Dist\text{-}InjExt}$ is NP-complete.*

PROOF: First of all, by 2.2.2, the problem is in NP. We know, from 1.4.10, that a given BPPO has an injective distributive extension iff it has a separating set of ideal binary decompositions. But we have shown in the preceding proposition that we can reduce the question of separability of a unit-free propositional formula to one of having a separating set of ideal binary decompositions in the corresponding BPPO. Note further that by 2.2.12, the height of this BPPO is 2, as is its meet fanout. Since Sat-Nonredundant-CNF is NP-complete, and since we can perform the translation from CNF formula to BPPO prespecification in polynomial time, we have that Dist-InjExt$(2, \infty, 2)$ is NP-complete. □

2.3 Refinement of NP-Complete Extension Results

In this subsection, we present a pair of refinements to the results of the previous section. First of all, we show that the problem Dist-InjExt remains NP-complete even if we restrict the join fanout to three. The critical result necessary to show this is to establish that we can preserve strong separability while translating into 3-CNF. Secondly, we show that the problems Dist-BinExt and Dist-InjExt both remain NP-complete even if we restrict the join fanout to 2, provided that we allow a height of 3.

We begin with a careful study of the translation from CNF to 3-CNF.

2.3.1 3-CNF translations Let φ be a CNF formula over the set $X = \{x_1, \ldots, x_n\}$ of variables. A *3-CNF translation* of φ, into a CNF formula with at most three literals per clause, is defined as follows. Suppose that φ may be written as $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_k$, with the φ_i's the clauses of φ, and with $\varphi_i = (\ell_{i1} \vee \ldots \vee \ell_{ik_i})$ the representation of the i^{th} clause in terms of its literals. We assume that all clauses, and all literals within a given clause, are distinct. We also have an implicit ordering on the clauses and on the literals within each clause, as reflected by the index set, but this choice of ordering is completely arbitrary. Now, to get a 3-CNF translation, replace each clause φ_i with the disjunction $\varphi_i^{(3)}$, defined as follows.

(i) If φ_i contains three or fewer literals, then let $\varphi_i^{(3)} = \varphi_i$.

(ii) If φ_i contains more than three literals, proceed as follows. Let $Z_i = \{z_{i1}, \ldots, z_{i(m_i-3)}\}$ be a set of $m_i - 3$ new variables, where m_i is the number of distinct literals in clause φ_i. Define

$$\begin{array}{rcl}
\varphi_i^{(3)} & = & (\ell_{i1} \vee \ell_{i2} \vee z_{i1}) \wedge \\
& & (\ell_{i3} \vee \neg z_{i1} \vee z_{i2}) \wedge \\
& & \vdots \\
& & (\ell_{ip} \vee \neg z_{i(p-2)} \vee z_{i(p-1)}) \wedge \\
& & (\ell_{i(p+1)} \vee \neg z_{i(p-1)} \vee z_{ip}) \wedge \\
& & \vdots \\
& & (\ell_{i(m_i-2)} \vee \neg z_{i(m_i-4)} \vee z_{i(m_i-3)}) \wedge \\
& & (\ell_{i(m_i-1)} \vee \ell_{im_i} \vee \neg z_{i(m_i-3)})
\end{array}$$

We always take $Z_i \cap Z_j = \emptyset$ for $i \neq j$, and $X \cap Z_i = \emptyset$ for all i. The set of variables for $\varphi^{(3)}$ is thus $X \cup (\bigcup_{i=1}^{k} Z_i)$, with $Z_i = \emptyset$ if φ_i has three or fewer literals.

This transformation is actually the standard one which is used to show that Sat-Exact-3CNF is NP-complete [AHU74, Thm. 10.4]. We have repeated the full construction here because we need to be very precise about how we label and handle the variables in the process of establishing strong separability.

We also define the following two useful functions.

$$\begin{aligned}\mathsf{Before}(z_{ip}) &= \{\ell_{it} \mid 1 \leq t \leq p+1\};\\ \mathsf{After}(z_{ip}) &= \{\ell_{it} \mid p+2 \leq t \leq m_i\}.\end{aligned}$$

Note that, for any clause φ_i which contains more than three literals, for any variable $z \in Z_i$, both both $\mathsf{Before}(z)$ and $\mathsf{After}(z)$ contain at least two literals.

As a notational convenience, to avoid repeating a massive amount of detail at each step, whenever we work with a CNF formula φ in that which follows, we shall assume that it has k clauses, and that the i^{th} clause contains m_i literals, as we have described above. We shall also assume that the decomposition has exactly the form described above, including the variable sets Z_i and the implicit (but arbitrary) ordering on the literals within the clauses.

Before proceeding, it is very useful to characterize just how models in the new 3-CNF formula relate to models in the original CNF formula. The following two results provide the necessary constraints. In each case, the proof follows easily upon examining the form of the 3-CNF transformation, and so is omitted.

2.3.2 Lemma *Let φ be a CNF formula over the set $X = \{x_1, \ldots, x_n\}$ of variables. Then a function $g : X \cup (\bigcup_{i=1}^{k}) \to \{\perp, \top\}$ is in $\mathsf{Mod}(\varphi^{(3)})$ iff the following conditions are satisfied.*

(i) *There is an $f \in \mathsf{Mod}(\varphi)$ such that for all $x \in X$, $f(x) = g(x)$.*

(ii) *For each $z_{ip} \in Z_i$:*

- *$g(z_{ip}) = \perp$ implies that there exists a variable $q \in \mathsf{Before}(z_{ip})$ with the property that $\bar{g}(\ell_{iq}) = \top$;*
- *$g(z_{ip}) = \top$ implies that there exists a variable $q \in \mathsf{After}(z_{ip})$ with the property that $\bar{g}(\ell_{iq}) = \top$.* $\square$

2.3.3 Proposition *Let φ be a CNF formula.*

(a) *φ is satisfiable iff $\varphi^{(3)}$ is.*

(b) *If $f \in \mathsf{Mod}(\varphi)$, then any $g : X \cup (\bigcup_{i=1}^{k} Z_i) \to \{\perp, \top\}$ satisfying the following conditions is in $\mathsf{Mod}(\varphi^{(3)})$.*

(i) *For all $x \in X$, $g(x) = f(x)$.*

(ii) *For each i such that φ_i has more than three literals, there is a nonempty subset $S \subseteq \{1, \ldots, m_i\}$ with $\bar{f}(\ell_{is}) = \top$ for each $s \in S$, with*

$$g(z_{ip}) = \begin{cases} \top & \text{if } p \leq s_1 - 2 \text{ or } p+2 \in S;\\ \perp & \text{otherwise,}\end{cases}$$

and s_1 denoting the least element of S. $\square$

We can now establish that strong separability is preserved under this transformation.

2.3.4 Lemma *Let φ be a nonredundant CNF formula. Then, if φ is strongly separable, so too is $\varphi^{(3)}$.*

PROOF: Assume that φ is strongly separable. Let X be the set of variables over which φ is taken. In view of the preceding proposition, it is clear that any two variables which are strongly separable in φ are also strongly separable in $\varphi^{(3)}$. We need to consider the cases involving the variables introduced in the 3-CNF transformation. Generally speaking, to give a complete description would require an incredible amount of tedious detail. Therefore, we show only the "critical" choices, and leave it to the reader to fill in the rest. The results of 2.3.2 and 2.3.3 are useful to keep in mind when doing this. First of all, we consider two "Z" variables associated with the same clause in φ; that is, a pair of the from $\{z_{ip}, z_{iq}\}$. Without loss of generality, assume that $p < q$. We have four cases to consider.

$[f(z_{ip}) = \bot; f(z_{iq}) = \bot]$: To find an $f \in \mathsf{Mod}(\varphi^{(3)})$ with this property, start by choosing a model $g \in \mathsf{Mod}(\varphi)$ with $\bar{g}(\ell_{i1}) = \top$. (This is possible because we are assuming that φ is strongly separable.) We build a model $f \in \varphi^{(3)}$ which is an extension of g. We may take $\bar{f}(z_{is}) = \bot$ for all s, $1 \leq s \leq m_i - 3$. This assignment to the z_{is}'s makes $\varphi_i^{(3)}$ true. For the other clauses, we simply choose assignments to the z_{rs}'s which make the entire formula true. In view of 2.3.2, this is always possible. Hence such an f exists in $\mathsf{Mod}(\varphi^{(3)})$.

$[f(z_{ip}) = \top; f(z_{iq}) = \top]$ This is similar to the previous case, except that we now choose $\bar{f}(\ell_{i(m_i)}) = \top$ and $\bar{f}(z_{is}) = \top$ for all s.

$[f(z_{ip}) = \bot; f(z_{iq}) = \top]$: Choose a $g \in \mathsf{Mod}(\varphi)$ with $\bar{g}(\ell_{i(p+1)}) = \top$ and $\bar{g}(\ell_{i(q+2)}) = \top$. This is possible since φ is nonredundant, and so $\ell_{i(p+1)}$ and $\ell_{i(q+2)}$ are literals of different variables. We extend g to an $f \in \mathsf{Mod}(\varphi^{(3)})$ with $f(z_{ip}) = \bot$ and $f(z_{iq}) = \top$ by choosing $S = \{p+1, q+2\}$ and applying part (b)(ii) of 2.3.3.

$[f(z_{ip}) = \top; f(z_{iq}) = \bot]$: Here we choose an $f \in \mathsf{Mod}(\varphi)$ with $\bar{f}(\ell_{iq}) = \top$. Then, letting $S = \{q\}$, we may apply (b)(ii) of 2.3.3 to get the required model.

Next, we consider the case in which two "Z" variables, z_{ip} and z_{jq}, are associated with different clauses of φ, φ_i and φ_j.

$[f(z_{ip}) = \bot; f(z_{jq}) = \bot]$: Since $\mathsf{After}(z_{ip})$ and $\mathsf{After}(z_{jq})$ each contain at least two literals, we may choose $\ell_{ir} \in \mathsf{After}(z_{ip})$ and $\ell_{js} \in \mathsf{After}(z_{jq})$, with $\mathrm{Var}(\ell_{ir}) \neq \mathrm{Var}(\ell_{js})$. Choose a model $g \in \mathsf{Mod}(\varphi)$ with $\bar{g}(\ell_{ir}) = \bar{g}(\ell_{js}) = \top$. Extend it to $f \in \mathsf{Mod}(\varphi^{(3)})$ with $f(z_{it}) = \top$ for $t \leq r-2$ and $f(z_{it}) = \bot$ otherwise. The rest of the extension is arbitrary, as long as it satisfies $\varphi^{(3)}$.

$[f(z_{ip}) = \top; f(z_{jq}) = \top]$: Similar, except use $\mathsf{Before}(z_{ip})$ and $\mathsf{Before}(z_{jq})$.

$[f(z_{ip}) = \bot; f(z_{jq}) = \top]$: Similar, except use $\mathsf{Before}(z_{ip})$ and $\mathsf{After}(z_{jq})$.

$[f(z_{ip}) = \top; f(z_{jq}) = \bot]$: Similar, except use $\mathsf{After}(z_{ip})$ and $\mathsf{Before}(z_{jq})$.

Finally, we must consider the case of a pair consisting of a variable x from X and a "Z" variable z_{jq}.

$[f(x) = \bot; f(z_{jq}) = \alpha]$: Here α is either $\bot$ or $\top$; we will treat both cases at the same time. Choose $\ell_{ir} \in \mathsf{Before}(z_{ip})$ with $\mathrm{Var}(\ell_{ir}) \neq x$. Pick $g \in \mathsf{Mod}(\varphi)$ with $g(x) = \alpha$ and $\bar{g}(\ell_{ir}) = \top$. Extend g to a model of $\varphi^{(3)}$ by choosing $f(z_{is}) = \top$ for $s \leq r-2$ and $f(z_i s) = \bot$. Then, $f(z_{ip}) = \bot$, as required.

$[f(x) = \bot; f(z_{jq}) = \alpha]$: Similar, except choose $\ell_{ir} \in \mathsf{After}(z_{ip})$ with $\mathrm{Var}(\ell_{ir}) \neq x$.

□

With this result in hand, the NP-completeness of the separability problems for logical formulas in 3-CNF follows easily.

2.3.5 Theorem — NP-completeness of separability testing restricted to 3-CNF *The problems* Sep-3CNF *and* StrongSep-3CNF *are NP-complete.*

PROOF: That these problems are in NP follows from 2.2.8, since 3-CNF formulas are a special case of CNF formulas. The NP-hardness follows from 2.3.4, proceeding exactly as in the proof of 2.2.8. □

2.3.6 Proposition *Let φ be a unit-free propositional formula. Then $\varphi^{(3)}$ is separable iff $\mathbf{S}_{\varphi^{(3)}}$ is consistent and its minimal extension $\mathsf{CanBPPO}(\mathbf{S}_{\varphi^{(3)}})$ to a BPPO has a separating set of ideal binary decompositions.*

PROOF: The proof is similar to that of 2.2.15. □

2.3.7 Theorem — NP-completeness of Dist-InjExt$(2,3,2)$ *The problem* Dist-InjExt$(2,3,2)$ *is NP-complete.*

PROOF: The proof goes as in 2.2.16. The only difference is that we now rely on the stronger results 2.3.4 and 2.3.6 instead of 2.2.6 and 2.2.8, and transform StrongSep-3CNF to Dist-InjExt$(2,3,2)$. The join fanout bound of 3 is an obvious consequence of the fact that we are using 3-CNF formulas. □

Thus, we have the "$(2,3,2)$" bound for the injective extension problem of a BPPO prespecification to a distributive lattice. We now turn to showing that we can maintain NP-completeness in the "$(3,2,2)$" context. That is, we can reduce the join fanout to 2 if we allow a height of 3.

2.3.8 (2,3)-hierarchical representations Let φ be a unit-free CNF formula. The *(2,3)-hierarchical representation* of φ, denoted $\varphi^{(2,3)}$, is defined as follows. Let $\varphi^{(3)}$ denote a formula equivalent to φ, constructed as described in 2.3.1. For each clause φ_i of φ, let $W_i = \{w_{i1}, \ldots, w_{i(m_i-2)}\}$ be a set of $m_i - 2$ new variables. In transforming φ to $\varphi^{(2,3)}$, we replace the clauses obtained from φ_i as follows.

(a) If φ_i contains fewer than three literals, then leave it as is.

(b) If $\varphi_i = (\ell_1 \vee \ell_i \vee \ell_3)$ contains exactly three literals, then replace it with the conjunction $(w_{i1} \Leftrightarrow (\ell_1 \vee \ell_2)) \wedge (w_{i1} \vee \ell_3)$.

(c) If φ_i contains more than three variables, then first transform it into a conjunction of clauses with at most three variables, as described in 2.3.1. Then, perform the following further replacements.

  (i) Replace $(\ell_{i1} \vee \ell_{i2} \vee z_{i1})$ with $(w_{i1} \Leftrightarrow (\ell_{i2} \vee z_{i1})) \wedge (w_{i1} \vee \ell_{i1})$.

  (ii) Replace each clause of the form $(\ell_{ip} \vee \neg z_{i(p-2)} \vee z_{i(p-1)})$ with $(w_{i(p-1)} \Leftrightarrow (\neg z_{i(p-2)} \vee z_{i(p-1)})) \wedge (w_{i(p-1)} \vee \ell_{ip})$.

  (iii) Replace $(\ell_{i(m_i-1)} \vee \ell_{im_i} \vee \neg z_{i(m_i-3)})$ with $(w_{i(m_i-2)} \Leftrightarrow (\neg \ell_{im_i} \vee \neg z_{i(m_i-3)})) \wedge (w_{i(m_i-2)} \vee \ell_{i(m_i-1)})$.

(d) If two of the w_{i1}'s are equivalent to exactly the same disjunction of literals, say $(w_{i1} \Leftrightarrow (\ell_1 \vee \ell_2)$ and $(w_{j1} \Leftrightarrow (\ell_1 \vee \ell_2)$, then replace each occurrence of w_{j1} with w_{i1}.

Figure 2.3 shows in pictures how this transformation behaves for a clause of the form $(\ell_1 \vee \ell_2 \vee \ell_3)$. The symbol $(\top)$ denotes a "virtual $\top$," as in Figure 2.2. Note that which literals are "combined" to become w depend upon the form of the clause. The rules identified above must be followed in the case that literals involving z_{ij}'s are involved.

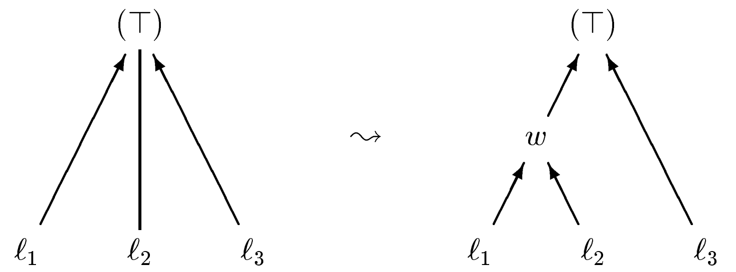

Figure 2.3: The translation from join fanout three to join fanout two.

If the original formula φ is strongly separable, then the resulting (2,3)-hierarchical formula is separable. The latter may not be strongly separable, but this is not a hindrance to proving the final NP-completeness result. Indeed, the reason that we established strong separability of $\varphi^{(3)}$ in 2.3.4, rather than just separability, was so that we could use it in the lemma below.

2.3.9 Lemma *Let φ be a nonredundant CNF formula.*

(a) $\varphi^{(2,3)}$ *is satisfiable iff* φ *is.*

(b) *If* φ *is strongly separable, then* $\varphi^{(2,3)}$ *is separable.*

PROOF: Part (a) is immediate, as these transformations clearly do not alter logical satisfiability. As for part (b), the result follows readily if we make full use of the fact that if φ is strongly separable, then so too is $\varphi^{(3)}$, as demonstrated in 2.3.4. We only need to ensure that we can separate the new "W" variables from each other, and from the other variables. In an equivalence of the form $w \Leftrightarrow (\ell_1 \vee \ell_2)$, we know that we can choose the values of ℓ_1 and ℓ_2 independently of each other, and so we can separate ℓ_1 from w by choosing one model in which ℓ_1 is false and ℓ_2 true, and then another in which both ℓ_1 and ℓ_2 are both false. Separability of w from the other variables is automatic, since they are strongly separable from the members of $\mathrm{Var}(\ell_1)$ and $\mathrm{Var}(\ell_2)$. To separate to "W" variables from each other, it suffices to observe that if we have $(w_i \Leftrightarrow (\ell_1 \vee \ell_2)$ and $(w_j \Leftrightarrow (\ell_3 \vee \ell_4)$, then in view of 2.3.8(d), we cannot have that $\{\ell_1, \ell_2\} = \{\ell_3, \ell_4\}$. Since the ℓ_i's are strongly separable from one another, it is easy to make a choice of pairs of models in which w_i and w_j will be separated. □

In analogy to 2.2.9, we define the BPPO corresponding to a (2,3)-hierarchical formula.

2.3.10 The BPPO prespecification of (2,3)-hierarchical formula Let φ be a unit-free CNF formula, and let $\varphi^{(2,3)}$ be its (2,3)-hierarchical representation. Assume the notation introduced in 2.3.8. Then the BPPO prespecification associated with $\varphi^{(2,3)}$, denoted $\mathbf{S}_{\varphi^{(2,3)}} = (P_{\varphi^{(2,3)}}, \mathbf{C}_{\varphi^{(2,3)}})$, is defined as follows. Define $V = X \cup Y \cup (\bigcup_{i=1}^{m_i-3} Z_i) \cup (\bigcup_{i=1}^{m_i-2} W_i)$. Define $P_{\varphi^{(2,3)}}$ and $\mathbf{C}_{\varphi^{(2,3)}}$ as follows.

(i) $P_{\varphi^{(2,3)}} = \mathsf{Literals}(V)$.

(ii) $\mathbf{C}_\varphi$ consists of the following rules.

- For each $v \in V$, $(\bigvee\{v, \neg v\} = \top)$, we have $(\bigwedge\{v, \neg v\} = \bot) \in \mathbf{C}_{\varphi^{(2,3)}}$.
- For each two-literal clause $(\ell_1 \vee \ell_2)$, we have $(\bigvee\{\ell_1, \ell_2\} = \top) \in \mathbf{C}_{\varphi^{(2,3)}}$.
- For each formula of the form $(v \Leftrightarrow (\ell_1 \vee \ell_2))$, we have $\bigvee(\{\ell_1, \ell_2\} = v) \in \mathbf{C}_{\varphi^{(2,3)}}$.

Once again, we have a result corresponding to 2.2.15. We omit the details of the proof, since they are entirely similar to those of 2.2.15.

2.3.11 Proposition *Let φ be a nonredundant propositional formula. Then $\varphi^{(2,3)}$ is separable iff $\mathbf{S}_{\varphi^{(2,3)}}$ is consistent and its minimal extension $\mathsf{CanBPPO}(\mathbf{S}_{\varphi^{(2,3)}})$ to a BPPO has a separating set of ideal binary decompositions.* □

Finally, we have the NP-completeness of the "(3,2,2)" problems.

2.3.12 Theorem NP-completeness of (3,2,2) problems *The problems* $\mathsf{Dist\text{-}BinExt}(3,2,2)$ *and* $\mathsf{Dist\text{-}InjExt}(3,2,2)$ *are NP-complete.*

PROOF: We already know that the problems are in NP. It is easy to see that, given a CNF formula φ, the height of $\mathbf{S}_{\varphi^{(2,3)}} = (P_{\varphi^{(2,3)}}, \mathbf{C}_{\varphi^{(2,3)}})$ is 3, and both the join and the meet fanouts are 2. The rest of the proof goes as in 2.2.14–2.2.16. The only difference is that we rely on the stronger results 2.3.9 and 2.3.11 to obtain the required separability. □

2.4 Cases Solvable in Deterministic Polynomial Time

In this section, we turn to the question of asking for which classes of finite BPPO prespecifications are extension questions to distributive structures solvable in deterministic polynomial time. To attack this problem, we need to "reverse" our perspective a bit. This time, rather than looking to transform satisfiability to extensibility, we seek to transform extensibility to satisfiability in polynomial time. If the resulting satisfiability problem is solvable in deterministic polynomial time, then so too will be the extensibility problem. To begin, then, we define the propositional formula associated with a prespecification.

2.4.1 The formula of a prespecification Let $\mathbf{S} = (P, \mathbf{C})$ be a consistent finite BPPO prespecification. The *formula* of $\mathbf{S}$, denoted $\varphi_{\mathbf{S}}$, is defined as follows.

(i) $\mathrm{Vars}(\varphi_{\mathbf{S}}) = P$.

(ii) $\varphi_{\mathbf{S}}$ is the conjunction of all formulas of the following form.

(a) For each $(\bigvee S = a) \in \mathbf{C}$ with $a \in P$, the formula $((\bigvee S) \Leftrightarrow a)$ is a conjunct of $\varphi_{\mathbf{S}}$. In the logical formula, $\bigvee$ denotes logical disjunction. Thus, if $S = \{s_1, \ldots, s_n\}$, then the formula is $((s_1 \vee \ldots \vee s_n) \Leftrightarrow a)$.

(b) For each $(\bigvee S = \top) \in \mathbf{C}$, the formula $(\bigvee S)$ is a conjunct of $\varphi_{\mathbf{S}}$. Thus, if $S = \{s_1, \ldots, s_n\}$, then the formula is $(s_1 \vee \ldots \vee s_n)$.

(c) For each $(\bigwedge S = a) \in \mathbf{C}$ with $a \in P$, the formula $((\bigwedge S) \Leftrightarrow a)$ is a conjunct of $\varphi_{\mathbf{S}}$. In the logical formula, $\bigvee$ denotes logical conjunction. Thus, if $S = \{s_1, \ldots, s_n\}$, then the formula is $((s_1 \wedge \ldots \wedge s_n) \Leftrightarrow a)$.

(d) For each $(\bigvee S = \top) \in \mathbf{C}$, the formula $(\bigwedge S)$ is a conjunct of $\varphi_{\mathbf{S}}$. Thus, if $S = \{s_1, \ldots, s_n\}$, then the formula is $(s_1 \wedge \ldots \wedge s_n)$.

(e) For each $(a < b) \in \mathbf{C}$, the formula $(a \Rightarrow b)$ is a conjunct of $\varphi_{\mathbf{S}}$.

Rules of the form $(\bigvee S = \bot)$ and $(\bigwedge S = \top)$ are not considered because they can never be part of a consistent prespecification.

Notice that we may effectively regard $\varphi_{\mathbf{S}}$ as being in CNF. Indeed, the formula $((s_1 \vee \ldots \vee s_n) \Leftrightarrow a)$ is equivalent to $((\neg s_1 \vee a) \wedge \ldots (\neg s_n \vee a)) \wedge (s_1 \vee \ldots \vee s_n \vee \neg a)$, and the formula $((s_1 \wedge \ldots \wedge s_n) \Leftrightarrow a)$ is equivalent to $((\neg a \vee s_1) \wedge \ldots \wedge (\neg a \vee s_n)) \wedge (a \vee \neg s_1 \vee \ldots \vee \neg s_n)$. We shall often implicitly adopt this convention in our subsequent arguments.

2.4.2 Example The ideas of the above construction are best illustrated by example. Let $\mathbf{S} = (P, \mathbf{C})$ be defined by $P = \{a, b, c, d\}$ and $\mathbf{C} = \{(\bigvee\{a, b, c\} = \top), (\bigwedge\{c, d\} = \bot), (\bigwedge\{b, d\} = a)\}$. Then

$$\begin{aligned} \varphi_{\mathbf{S}} &= (a \vee b \vee c) \wedge (\neg c \vee \neg d) \wedge ((b \wedge d) \Leftrightarrow a)) \\ &\equiv (a \vee b \vee c) \wedge (\neg c \vee \neg d) \wedge (\neg b \vee \neg d \vee a) \wedge (\neg a \vee b) \wedge (\neg a \vee d) \end{aligned}$$

Because the variables of $\varphi_{\mathbf{S}}$ are exactly the inner types of $\mathbf{S}$, it follows immediately that satisfiability of $\varphi_{\mathbf{S}}$ corresponds to somewhere separability of $\mathbf{S}$, and that separability, Q-separability, and strong separability of $\varphi_{\mathbf{S}}$ correspond to the concepts of the same name in $\mathbf{S}$. We record these facts in a formal lemma.

2.4.3 Lemma *Let $\mathbf{S} = (P, \mathbf{C})$ be a consistent finite BPPO prespecification.*

(a) $\mathbf{S}$ *is somewhere separable iff it $\varphi_{\mathbf{S}}$ is satisfiable.*

(b) $\mathbf{S}$ *is separable (resp. strongly separable) iff $\varphi_{\mathbf{S}}$ is.* □

(c) *Given a set Q of pairs of elements from $\mathsf{Aug}(P)$, $\mathbf{S}$ is Q-separable iff $\varphi_{\mathbf{S}}$ is.* □

Testing for separability of formulas involves testing for satisfiability under conditions in which the truth values of certain variables are held fixed. We formalize this idea with the idea of binding of variables in propositional formulas.

2.4.4 Binding of propositional formulas

(a) Let φ be a propositional formula taken over a set X of variables, and let $B = \{\ell_1, \ldots, \ell_k\}$ be a set of literals over X. The formula $\varphi[B]$ is defined to be $\varphi \wedge (\ell_1 \wedge \ldots \wedge \ell_k)$, and is called the *binding of φ by B.*

(b) Let $\mathcal{F}$ be a class of propositional formulas. We say that $\mathcal{F}$ is *tractable for bindings* if there is a polynomial of one variable p such that for every formula $\varphi \in \mathcal{F}$ and every set of bindings B, there is a formula $\psi \in \mathcal{F}$ which is logically equivalent to $\varphi[B]$, and which is computable by a function in $O(p(\text{length}(\varphi)))$.

2.4.5 Examples Almost any "reasonable" class of formulas is tractable for bindings. Simple examples include the CNF formulas, the k-CNF formulas for any $k \geq 1$, and the Horn formulas. Indeed, the idea behind a binding is to "fix" the value of one or more variables, and we can typically find a *shorter* formula which reflects the binding, so that $p(x)$ may be taken to be the identity polynomial x in the above definition. For example, if we start with a CNF formula φ, and wish to find an equivalent formula to $\varphi[\ell]$ for some literal ℓ, we simply delete all clauses in which ℓ appears (since they will be true under the binding in any case), and remove all instances of the complement of ℓ from all remaining clauses (since that complementary literal cannot be true).

We are particularly interested in classes of formulas which are not only tractable for bindings, but also admit deterministic polynomial-time testing for satisfiability. The reason is reflected in the following lemma.

2.4.6 Lemma *Let $\mathcal{F}$ be a class of propositional formulas which is closed under conjunction and is tractable for bindings. Then, if satisfiability is solvable in deterministic polynomial time for formulas in $\mathcal{F}$, so too are the following questions:*

(a) *Given a formula φ over the set X of variables and a set Q of pairs of elements of $X \cup \{\bot, \top\}$, the question of whether or not φ has a Q-separator.*

(b) *Given a formula φ over the set X of variables and a set $S \subseteq P$, the question of whether or not there is a set $\Delta \subseteq \mathsf{Mod}(\varphi)$ which renders S atomic.*

(c) *The question of whether or not φ has a strong separator.*

PROOF: If φ is not satisfiable, then the answer to each of the above questions is false. Therefore, in the following, we assume that φ is satisfiable. (a) Let φ be a formula in $\mathcal{F}$. To test for separability of a pair of the form $\{x, \bot\}$ with $x \in \mathrm{Vars}(\varphi)$, we simply test $\varphi[\{x\}]$ for satisfiability. To test for separability of a pair of the form $\{x, \top\}$, we test $\varphi[\{\neg x\}]$ for satisfiability. To test a pair of the form $\{x, y\}$ for separability with $x, y \in \mathrm{Vars}(\varphi)$, we need to verify that at least one of the formulas in $\{\varphi[\{x, \neg y\}], \varphi[\{\neg x, y\}]\}$ is satisfiable, and at least one of the formulas in $\{\varphi[\{x, y\}], \varphi[\{\neg x, \neg y\}]\}$ is satisfiable. If even one of the variables in $\{x, y\}$ is not in $\mathrm{Vars}(\varphi)$, then the pair is automatically separable. Thus, a test for separability of a given pair $\{x, y\}$ may be performed in polynomial time. Now if φ is taken over the set X of variables, and $n = \mathsf{Card}(\mathrm{Vars}(\varphi))$, then any set Q of pairs of elements from $X \cup \{\bot, \top\}$ can have at most $(n+2) \cdot (n+1)$ pairs in which both elements are in $\mathrm{Vars}(\varphi) \cup \{\bot, \top\}$, which are the only pairs for which the test is nontrivial. Thus, we need perform at most $O(n^2)$ such tests, each of which may be performed in deterministic polynomial time. Hence the whole test may be performed in deterministic polynomial time.

(b) Let φ be a formula in $\mathcal{F}$, and let $S \subseteq \mathrm{Vars}(\varphi)$. Assume, without loss of generality, that $\mathrm{Vars}(\varphi) = X = \{x_1, \ldots, x_n\}$, let $k = \mathsf{Card}(S)$, and introduce k new sets of variables $X_j = \{x_{1j}, \ldots, x_{nj}\}$, $1 \leq j \leq k$. Let φ_j be the formula which is the same as φ, except that for each i, $1 \leq i \leq n$, x_{ij} has been substituted for x_i. Let $\psi = (\bigwedge_{j=1}^{n} \varphi_j)$. In other words, each conjunct has its own set of variables, with those of φ_j carrying j as a second subscript. Finally, let $f : \{1, \ldots, k\} \to S$ be any bijection, which will be used just for naming. Now, for $1 \leq i, j \leq k$, let

$$\ell_{ij} = \begin{cases} f(i)_j & \text{if } i = j; \\ \neg f(i)_j & \text{if } i \neq j; \end{cases}$$

and define $B = \{\ell_{ij} \mid 1 \leq i \leq k \text{ and } 1 \leq j \leq n\}$. Some further clarification of the notation is in order. For $1 \leq i \leq k$, note that $f(i)$ is a variable in S. Thus, $f(i)_j$ is a subscripted variable in X, which is effectively a variable in X_j. For $i = j$, the literal $\ell_{ij} = f(i)_j$ asserts that, in the j^{th} submodel φ_j, the variable $f(i)$ (represented by indexing as $f(i)_j$) is true. For $i \neq j$, the literal $\ell_{ij} = \neg f(i)_j$ asserts that the variable $f(i)$ (represented as $f(i)_j$) is false in the j^{th} model. Then, a model for $\psi[B]$ will effectively be k models for φ, with the j^{th} model obtained by looking only at the variables in X_j. Furthermore, it will only be the j^{th} model which will have $f(j) = \top$. (That is, $f(j)_j = \top$, and $f(j)_m = \bot$ for all $m \neq j$.) Hence, ψ is satisfiable iff there is a set of models which renders S atomic. The size of ψ is bounded by $\mathsf{Size}(\varphi) \cdot \mathsf{Card}(S)$, which is certainly polynomial in the size of the input. The satisfiability check may be performed in deterministic polynomial time, by assumption.

(c) To test φ for strong separability, we run four tests for each pair of variables from $\mathrm{Vars}(\varphi)$. Namely, for each such pair $\{x, y\}$, we test $\varphi[\{x, y\}]$, $\varphi[\{\neg x, y\}]$, $\varphi[\{x, \neg y\}]$, and $\varphi[\{\neg x, \neg y\}]$ for satisfiability. For a pair in which only one of the two variables (say x) is in $\mathrm{Vars}(\varphi)$, we run only two satisfiability tests, one on $\varphi[\{y\}]$ and one on $\varphi[\{\neg y\}]$. Again, it is easy to see that the whole collection of tests may be performed in deterministic polynomial time. □

We can now show that if we restrict height, join fanout, and meet fanout each to at most two, then we obtain a class of extension problems which are solvable in deterministic polynomial time.

2.4.7 Theorem – Extension problems solvable in deterministic polynomial time *The problems* Dist-BinExt$(2, 2, 2)$ *and* Dist-InjExt$(2, 2, 2)$ *are each solvable in deterministic polynomial time.*

PROOF: The key is to observe that if a prespecification $\mathbf{S} = (P, \mathbf{C})$ is of height 2, then it can only have rules in $\mathbf{C}$ of the form identified in 2.4.1 parts (ii)(b), (d), and (e). In other words, rules of the form $(\bigvee S = x)$ and $(\bigwedge S = x)$ are not possible unless x is $\top$ in the first case and $\bot$ in the second. The fanout bound of 2 implies that S cannot have more than two elements in such a rule. Thus, the conjuncts of $\varphi_{\mathbf{S}}$ are clauses with two literals each; *i.e.*, they are 2-CNF formulas. However, the problem Sat-2CNF is known to be solvable in deterministic polynomial time [EAS76]. From 2.4.6, we conclude that the tests for satisfiability and separability each may be performed in deterministic polynomial time. But then 2.4.3 tells us that somewhere separability and full separability of the prespecification itself is testable in deterministic polynomial time. Finally, the characterizations of 1.4.10 show us that the problems Dist-BinExt$(2, 2, 2)$ and Dist-InjExt$(2, 2, 2)$ must be solvable in deterministic polynomial time as well. □

There is another situation in which we can establish that the distributive extension problem is solvable in deterministic polynomial time – namely, when the prespecification has only join rules or only meet rules. The decision problem is effectively "trivial" in this case, once we have decided consistency of the prespecification.

2.4.8 One-way prespecifications Let $\mathbf{S} = (P, \mathbf{C})$ be a finite BPPO prespecification. We say that $\mathbf{S}$ is a *join-only prespecification* if $\mathbf{C}$ contains no nontrivial meet rules. Dually, $\mathbf{S}$ is a *meet-only prespecification* if $\mathbf{C}$ contains no nontrivial join rules. $\mathbf{S}$ is said to be *one-way* if it is either join only or else meet only.

2.4.9 Lemma *Let* $\mathbf{S} = (P, \mathbf{C})$ *be a consistent one-way finite BPPO prespecification. Then* $\mathbf{S}$ *is fully separable.*

PROOF: We start by assuming that $\mathbf{S}$ is join-only. Suppose that $\mathbf{S}$ is consistent. Then, with respect to the order $\preceq_{\mathbf{S}}$, we have by 2.1.7 and 2.1.8 that every principal preideal with respect to $\preceq_{\mathbf{S}}$ is an ideal of $\mathbf{S}$. Since $\mathbf{S}$ is join-only, it is clear that the function $f_\top : P \to \{\bot, \top\}$ defined by $f_\top(a) = \top$ for all $a \in P$ is a model of $\varphi_{\mathbf{S}}$. On the other hand, for each $a \in P$, the function $f_a : P \to \{\bot, \top\}$ defined by

$$f_a(x) = \begin{cases} \bot & \text{if } x \in \mathsf{PrinIdeal}(a, \preceq_{\mathbf{S}}); \\ \top & \text{otherwise}; \end{cases}$$

is a model of $\varphi_{\mathbf{S}}$. But if $a, b \in P$ with $a \neq b$, then we cannot have $\mathsf{PrinIdeal}(a, \preceq_{\mathbf{S}})$ $= \mathsf{PrinIdeal}(b, \preceq_{\mathbf{S}})$, since $\preceq_{\mathbf{S}}$ is a partial order, and hence antisymmetric. Thus, we must have that either $f_a(b) = \bot$, or else that $f_b(a) = \bot$. Since we always have $f_a(a) = f_b(b) = \top$, it follows that either f_a or else f_b has the property that its truth value on a is different than its truth value on b. Thus, at least one of $\{f_\top, f_a\}$ and $\{f_\top, f_b\}$ must be a separator for the pair $\{a, b\}$, so that any two elements of P are separable. Since $f_\top$ is a model, $\{f_\top\}$ is a separator for $\{a, \bot\}$ for any $a \in P$. Finally, to show that a pair of the form $\{a, \top\}$ is separable, it suffices to note that $\top$ is only in the principal ideal generated by itself. Thus, $\{f_a\}$, as defined above, is a separator for $\{a, \top\}$. We can thus invoke 2.4.3(b) to show that $\mathbf{S}$ is separable.

The proof for a meet-only prespecification is entirely dual to the one given above. □

We thus have that testing for distributive extensions of prespecifications with only join rules or only meet rules (in addition to partial-order rules), consistency is all that need be verified.

2.4.10 Theorem – another extension problem solvable in deterministic polynomial time *The problems* $\mathsf{Dist\text{-}BinExt}(\infty, \infty, 1)$ *and* $\mathsf{Dist\text{-}InjExt}(\infty, 1, \infty)$ *are each solvable in deterministic polynomial time.*

PROOF: Since $\mathsf{PreBPPO\text{-}Consis}$ is solvable in deterministic polynomial time (see 2.1.17(c)), this follow immediately from the previous lemma. □

2.4.11 Comparison to ALE-like systems The situation in which we have no join rules is much closer to that of systems such as ALE. However, our formalism is still somewhat more general, in that we allow order relationships such as $a \leq b$ and $a \leq c$ without requiring that $\bigwedge\{b, c\} = a$. Nonetheless, the great reduction in computational complexity which is achieved by disallowing join rules is evident. Indeed, it is easy to see that for such a prespecification $\mathbf{S}$, the resulting formula $\varphi_{\mathbf{S}}$ is a Horn formula. This essentially means that all inferences may be performed in deterministic time linear in the size of $\varphi_{\mathbf{S}}$ [DG84]. It also means that the algorithms for consistency checking sketched in Section 2.1 may be simplified substantially, although we do not develop the details here. The point to be made here is that the major reason that the distributive extension problems that we examined in Sections 2.2 and 2.3 are NP-complete is that both joins and meets of types are allowed. Allow only one, as in ALE, LKB, or LIFE, and the complexity dips tremendously. Roughly, it reduces to a comparison between the complexity of satisfiability of arbitrary formulas versus satisfiability of Horn formulas.

2.5 Special Computational Problems

In this section, we take a look at the computational problems involved in adding some stronger constraints to finite BPPO prespecifications, particularly statements that certain elements must be atoms. (Section 1.5 contains the basic definitions and results about atoms within our context.) We also take a look at how Q-separability affects the computational complexity of various problems.

2.5.1 Computational problems involving atoms We begin by identifying some fundamental decision problems involving atoms. Basically, the problems identified in 2.2.1 are expanded to allow declarations of atoms as well.

The decision problem $\mathsf{Atom\text{-}Dist\text{-}BinExt}(h, j, m)$:

> **Input:** A BPPO prespecification $\mathbf{S} = (P, \mathbf{C})$, with height bounded by h, $\bigvee$-fanout bounded by j, and $\bigwedge$-fanout bounded by m, and a subset $S \subseteq P$.
>
> **Question:** Does $\mathbf{S}$ have a consistent extension to a BPPO, which is atomic for S, to any of the distributive extension categories?

The decision problem $\mathsf{Atom\text{-}Dist\text{-}InjExt}(h, j, m$:

> **Input:** A BPPO prespecification $\mathbf{S} = (P, \mathbf{C})$, with height bounded by h, $\bigvee$-fanout bounded by j, and $\bigwedge$-fanout bounded by m, and a subset $S \subseteq P$.
>
> **Question:** Does $\mathbf{S}$ have a full separator which also renders S atomic?

The problems which were NP-complete without declarations of atoms remain NP-complete even with these declarations. The formalization and proof are as follows.

2.5.2 Theorem — NP-completeness *The problems* $\mathsf{Atom\text{-}Dist\text{-}BinExt}(h, j, m)$ *and* $\mathsf{Atom\text{-}Dist\text{-}InjExt}(h, j, m)$ *are NP-complete for* $(h, j, m) = (2, 3, 2)$ *and for* $(3, 2, 2)$.

PROOF: The NP-hardness of these problems is immediate, because $\mathsf{Dist\text{-}BinExt}(i, j, m)$ is a special case of $\mathsf{Atom\text{-}Dist\text{-}BinExt}(i, j, m)$ and $\mathsf{Dist\text{-}InjExt}(i, j, m)$ is a special case of $\mathsf{Atom\text{-}Dist\text{-}InjExt}(i, j, m)$. We may thus refer to 2.3.7 and 2.3.12 and be done. To show that the problems are in NP, we proceed essentially as in 2.2.2. The only difference is that it must be checked whether a candidate solution Δ also renders the proposed atom set S atomic. But this can clearly be done in time linear in the size of Δ. $\square$

When we take a problem which was previously solvable in deterministic polynomial time and add atomic constraints, things may or may not become more complex, depending upon the problem. In the "(2,2,2)" case, if the previous problem asked only for the existence of a nontrivial extension, without any injectivity constraint, then we continue to have a problem solvable in deterministic polynomial time.

2.5.3 Theorem — polynomial time solvability *The problem* $\mathsf{Atom\text{-}Dist\text{-}BinExt}(2,2,2)$ *is solvable in deterministic polynomial time.*

PROOF: Let $\mathbf{S} = (P, \mathbf{C})$ be a finite BPPO prespecification, and let $S \subseteq P$. If $S = \emptyset$, then the problem reduces to $\mathsf{Dist\text{-}BinExt}(2,2,2)$, which we know is solvable in deterministic polynomial time by 2.4.7. If S is nonempty, then any set Δ which renders S atomic must contain at least one ideal binary decomposition. But this means that $\mathbf{P}$ has a distributive extension pair, by 1.4.7. Thus, it suffices to show that we can find such a Δ in deterministic polynomial time. To see how this is done, let $S = \{s_1, \ldots, s_n\}$, and for $1 \leq i \leq n$, and let $S^{(i)} = \{s_1, \ldots, s_{i-1}, \neg s_i, s_{i+1}, \ldots s_n\}$. Then there exists a Δ which renders S atomic iff each of the formulas $\varphi[S^{(i)}]$, $1 \leq i \leq n$ is satisfiable. In view of 2.4.6, this question may be decided in deterministic polynomial time, whence the result. $\Box$

On the contrary, if we try to combine injectivity with the declaration of atoms, things appear to be much more complex. We do not provide a formal analysis, but the following discussion will provide some idea of why this is the case.

2.5.4 Discussion of the problem $\mathsf{Atom\text{-}Dist\text{-}InjExt}(2,2,2)$ One might conjecture at first that, in view of 2.4.7, that $\mathsf{Atom\text{-}Dist\text{-}InjExt}(2,2,2)$ is solvable in deterministic polynomial time as well. However, it does not appear that this is the case. Of course, it is certainly in NP, as it is a special case of the problems shown to be NP-complete in the theorem above. Now, given a formula φ and a set $S \subseteq \mathrm{Vars}(\varphi)$, we may use 2.4.6 to show that the individual test of full separability and whether or not there is a set of models which renders S atomic may each *separately* be solved in deterministic polynomial time. However, they do not appear to combine to yield a polynomial time solution. The difficulty arises because requiring a set $\Delta \subseteq \mathsf{Mod}(\varphi)$ to render S atomic places restrictions on the collection of models, whereas the process of building a full separator simply combines models. Thus, a separator may be constructed sequentially, one pair at a time, so adding models to Δ does not destroy the property of being a separator. However, the property of rendering S atomic is not preserved under adding new models. Therefore, the two algorithms cannot simply be combined. The results found in [GJS76] suggest that the problem is in fact NP-complete, but we do not pursue this question further in this report.

If we look at type hierarchies defined only by meet rules, then adding declarations of atoms is effectively trivial, as the following result shows.

2.5.5 Theorem — further deterministic polynomial time case *The problems* $\mathsf{Atom\text{-}Dist\text{-}BinExt}(\infty,1,\infty)$ *and* $\mathsf{Atom\text{-}Dist\text{-}InjExt}(\infty,1,\infty)$ *is solvable in deterministic polynomial time.*

PROOF: This is very easy; we simply apply the construction described in Section 0.2. The atoms in this construction are just the elements in the BPPO prespecification for which $\perp$ is the only lesser element. No elements are coalesced, so the extension is injective. $\Box$

On the other hand, type hierarchies involving only join rules do not interact cleanly with declarations of atoms. The following discussion gives some insights into the problems involved.

2.5.6 Discussion of the problems $\mathsf{Atom\text{-}Dist\text{-}BinExt}(\infty,\infty,1)$ **and** $\mathsf{Atom\text{-}Dist\text{-}InjExt}(\infty,\infty,1)$ We cannot use the "duality" between the $(\infty,1,\infty)$ and $(\infty,\infty,1)$ cases, as we did in 2.4.9 and 2.4.10. This is because in a bounded lattice the dual of an atom is not an atom, but rather a *co-atom*; that is, an element which has no greater element other than $\top$. In fact, even with a BPPO prespecification with a meet fanout of one, the addition of more than one atom guarantees that there is a nontrivial meet constraint. This is because for each pair of distinct atoms $\{a,b\}$, we must have $\bigwedge\{a,b\} = \bot$. For this reason, we conjecture that the problems $\mathsf{Atom\text{-}Dist\text{-}BinExt}(\infty,\infty,1)$ and $\mathsf{Atom\text{-}Dist\text{-}InjExt}(\infty.\infty,1)$ are NP-complete. However, the techniques which we have developed in this report do not appear to be directly applicable to this problem, and so we leave it open.

We now turn to a brief examination of the issues surrounding Q-separability. This is an important question because a user of a system may not always want or need all declared types to be separable. Rather, it may be essential to separate only certain identified types. We now show that problems of such Q-separability have the same complexity (up to deterministic polynomial equivalence) as problems of full separability. We being by formalizing the decision problems under investigation.

2.5.7 Questions involving Q-separability

The decision problem $\mathsf{GenDistProb}(h,j,m)$:

> **Input:** A BPPO prespecification $\mathbf{S} = (P,\mathbf{C})$, with height bounded by h, $\bigvee$-fanout bounded by j, and $\bigwedge$-fanout bounded by m, and a set Q of pairs of elements, each in $P \cup \{\bot,\top\}$.
>
> **Question:** Does $\mathbf{S}$ have a consistent extension to a BPPO, which in turn has an Q-injective extension, which is atomic for S, to any of the distributive extension categories?

2.5.8 Theorem — complexity results for $\mathsf{GenDistProb}$ *The problem* $\mathsf{GenDistProb}(h,j,m)$ *is NP-complete for* $(h,j,m) = (2,3,2)$ *or* $(3,2,2)$*, and is solvable in deterministic polynomial time for* $(h,j,m) = (2,2,2)$*,* $(\infty,1,\infty)$*, and* $(\infty,\infty,1)$*.*

PROOF: For the (2,3,2) and (3,2,2) cases, this follows immediately from 2.3.7 and 2.3.12, since Q-separability is a generalization of full separability. (The NP membership of these problems is proven in the same way as in 2.2.2). The deterministic polynomial-time cases are handled as follows. For the (2,2,2) case, we proceed exactly as in 2.4.7, save that we use 2.4.6(a) and 2.4.3(c) as the supporting arguments. For the $(\infty,1,\infty)$ and $(\infty,\infty,1)$ cases, the proof of 2.4.10 works as well, since full separability implies Q-separability for any Q. □

2.5.9 Remarks on combining atoms and Q-separability Since full separability is a special case of Q-separability, in view of the previous result, it should be clear that combining questions of rendering a set S atomic and providing Q separability does not introduce any radically new questions. The results in 2.5.2–2.5.6 could have injectivity replaced with Q-separability without any change in the identified complexities.

3. Conclusions and Further Directions

In this report, we have attempted to complement the practical work of the CUF system by laying the theoretical foundations for the partially specified type hierarchies which are central to that system. Specifically, we have accomplished the following.

(a) We have formalized the framework — bounded posets with partial orders (BPPO's) — in which partially specified type hierarchies may be systematically studied.

(b) We have systematized the various types of extensions, to a complete hierarchy, that a BPPO may have. Any such reasonable extension must be to a bounded distributive lattice, but there are many variations within this theme.

(c) We have studied the impact that declarations of constant types — types which cannot have nontrivial subtypes — have upon the nature of distributive extensions. Interestingly, without constant declarations, in all cases which we consider, there is always a "natural" extension, which does not involve any nondeterministic choice in the form of the extension. However, in the presence of constant declarations, this property disappears, and a choice between several alternatives may need to be made. Thus, the addition of constant declarations to a type declaration system has significant implications in terms of nondeterminism.

(d) We have established a close connection between certain decision problems in propositional logic and the decision problems connected with determining whether or not a specification has an extension of a certain kind. We then use this connection to establish results on the computational complexity of deciding whether or not certain kinds of distributive extensions exist. Using the height (length of the longest path in the specification) and fanout (the number of elements in a join or meet specification), we show that even under extremely tight constraints (one of these parameters equal to three and the other two equal to two), all of the extension decision problems are NP-complete. It is only when we drop down to relatively trivial cases that we find problems which are soluble in deterministic polynomial time.

As future directions, the following seem the most useful.

(1) As is the case with most NP-complete problems, the extension problems studied here need to be examined from angles which will yield efficient algorithms for a useful variety of "practical" cases. Given the close connection

between extensibility and satisfiability of certain associated formulas, one obvious tact would be to examine the relevance of characterizing formulas which admit efficient tests for satisfiability, thus extending the rather basic results of 2.4.7, 2.4.10, and 2.5.3. One possibility is to examine the relevance of classes of so-called *generalized Horn formulas* [Asp80], [YD83], [CCH*90], [Bun93]. These formulas all admit polynomial-time decision algorithms for satisfiability. However, it is not clear that they provide a useful basis for the kind of constraint specifications the arise in practice; this will need to be studied more closely. Another tactic might be to note that real hierarchies tend to be rather simple in their "intertwining" of the relationships between the types. Perhaps by limiting the interaction of types in some way in accord with typical use, tractable algorithms may be obtained.

(2) The type hierarchies of feature-based systems such as CUF have features attached to the types. CUF is particularly sophisticated in that it admits polymorphic feature assignment — the same feature may be assigned to different types, with different semantics. It is a critical next step in this research to integrate feature specification into our framework. Particularly, it is critical to examine what effect this integration would have on determining which types *must* have distinct instantiations, and which may be collapsed to a common type.

(3) Often, to obtain a distributive hierarchy, it becomes mandatory to require that two types have identical instantiations (as in the case of a diamond or pentagon in the specification). Currently, the CUF system does not require types with distinct names to have distinct instantiations, unless the feature declarations mandate this. CUF will not inform the user that two types must have identical instantiations. It would clearly be a useful addition to the CUF system to provide the user with information on which types must coincide for the hierarchy to be distributive. Furthermore, it would be useful to allow the user to specify, independently of any feature declarations, that two types must be distinct. In short, it would be most useful if the user were to know just how the type hierarchy is realized. Our theory provides the basis for the study of this problem.

(4) It is clear from our results that allowing constant declarations complicates matters substantially, in that "natural" extensions no longer exist and arbitrary choices must be made. In practical terms, this means that there will be several disjunctive alternatives which must be considered. With a more thorough understanding of just how such alternatives arise, a substantial saving in the complexity of the algorithms which consider, in turn, the alternatives, may be made. Particularly, in the absence of constant declarations, there need be no nondeterminism in the collapsing of the hierarchy. Therefore, we believe that a more thorough study of how the disjunctive alternatives are identified and presented, in the light of our results, could result in a more informative and efficient type management system for CUF-like systems.

4. Acknowledgments

This work was completed while the author was a research fellow, funded by the Norwegian Research Council, at the Department of Informatics of the University of Bergen.

Tore Langholm, Chris Mellish, and Richard Moe all read parts of this report and made valuable comments, which have (hopefully) led to an improved presentation of the ideas. Naturally, any remaining errors are the responsibility of the author.

References

[AHU74] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Ait93] Aït-Kaci, H., "An introduction to life – programming with logic, inheritance, functions, and equations," in: Miller, D., ed., *Proceedings of the International Symposium on Logic Programming, (Vancouver, Canada)*, pp. 52–68, 1993.

[AN86] Aït-Kaci, H. and Nasr, R., "LOGIN: A logic programming language with built-in inheritance," *J. Logic Programming*, **3**(1986), pp. 185–215.

[AP93] Aït-Kaci, H. and Podelski, A., "Towards a meaning of LIFE," *J. Logic Programming*, **16**(1993), pp. 195–234.

[Asp80] Aspvall, B., "Recognizing disguised NR(1) instances of the satisfiability problem," *J. Algorithms*, **1**(1980), pp. 97–103.

[Bir67] Birkhoff, G., *Lattice Theory, third edition*, American Mathematical Society, 1967.

[BdPC93] Briscoe, T., de Paiva, V., and Copestake, A., eds., *Inheritance, Defaults, and the Lexicon*, Cambridge University Press, 1993.

[Bun93] Bünning, H. K., "On generalized Horn formulas and k-resolution," *Theoret. Comput. Sci.*, **116**(1993), pp. 405–413.

[Car92a] Carpenter, B., ALE: The Attribute Logic Engine user's guide, Technical Report, Carnegie Mellon University, Laboratory for Computational Linguistics, 1992.

[Car92b] Carpenter, B., *The Logic of Typed Feature Structures*, Cambridge University Press, 1992.

[CCH*90] Chandru, V., Coullard, C. R., Hammer, P. L., Montañez, M., and Sun, X., "On renamable Horn and generalized Horn functions," *Ann. Math. Art. Intell.*, **1**(1990), pp. 33–47.

[CLR90] Cormen, T. H., Leiserson, C. E., and Rivest, R., *Introduction to Algorithms*, MIT Press, 1990.

[DP90] Davey, B. A. and Priestly, H. A., *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[DD93] Döerre, J. and Dorna, M., "CUF – a formalism for linguistic knowledge representation," in: Dörre, J., ed., *Computational Aspects of Constraint-Based Linguistic Description, DYANA-2 Deliverable R.1.2.A*, pp. 3–22, ESPRIT, 1993.

[Dor94] Dorna, M., The CUF user's manual, Technical Report, Institut für maschinelle Sprachverarbeitung, Universität Stuttgart, 1994.

[DG84] Dowling, W. F. and Gallier, J. H., "Linear-time algorithms for testing the satisfiability of propositional Horn clauses," *J. Logic Programming*, **3**(1984), pp. 267–284.

[EAS76] Even, S., Atai, A., and Shamir, A., "On the complexity of timetable and multicommodity flow problems," *SIAM J. Computing*, **5**(1976), pp. 691–703.

[GJ79] Garey, M. R. and Johnson, D. S., *Computers and Intractability*, W. H. Freeman, 1979.

[GJS76] Garey, M. R., Johnson, D. S., and Stockmeyer, L., "Some simplified NP-complete graph problems," *Theoret. Comput. Sci.*, **1**(1976), pp. 237–267.

[Gra68] Grätzer, G., *Universal Algebra*, D. Van Nostrand, 1968.

[Gra78] Grätzer, G., *General Lattice Theory*, Academic Press, 1978.

[HS73] Herrlich, H. and Strecker, G. E., *Category Theory*, Allyn and Bacon, 1973.

[Joh88] Johnson, M., *Attribute-Value Logic and the Theory of Grammar*, CSLI, 1988.

[Kap89] Kaplan, R. M., "The formal architecture of Lexical-Functional Grammar," in: *Proceedings of ROCLING II*, pp. 1–18, 1989.

[LB87] Levesque, H. J. and Brachman, R. J., "Expressiveness and tractability in knowledge representation," *Computational Intelligence*, **3**(1987), pp. 78–93.

[Man93] Manandhar, S., "CUF in context," in: Dörre, J., ed., *Computational Aspects of Constraint-Based Linguistic Description, DYANA-2 Deliverable R.1.2.A*, pp. 45–53, ESPRIT, 1993.

[PS87] Pollard, C. and Sag, I. A., *Information-Based Syntax and Semantics, Volume 1: Fundamentals* of *CSLI Lecture Notes No. 13*, CSLI, 1987.

[Shi86] Shieber, S. M., *An Introduction to Unification-Based Approaches to Grammar*, University of Chicago Press, 1986.

[Shi92] Shieber, S. M., *Constraint-Based Grammar Formalisms*, MIT Press, 1992.

[Wal91] Walters, R. F. C., *Categories and Computer Science*, Cambridge University Press, 1991.

[YD83] Yamasaki, S. and Doshita, S., "The satisfiability problem for a class consisting of Horn clauses and some non-Horn sentences in propositional logic," *Inform. and Control*, **59**(1983), pp. 1–12.

[Zaj91] Zajac, R., Notes on the Typed Feature System, Version 4, January 1991, Technical Report, Universität Stuttgart, Institut für Informatik, Project Polygloss, 1991.

[Zaj92] Zajac, R., "Inheritance and constraint-based grammar formalisms," *Computational Linguistics*, **18**(1992), pp. 159–182.

Appendix. Relationship to the CUF Formalism

The purpose of this appendix is to sketch how the formal results which were presented in this report are related to the more practical feature-structure programming language CUF. It is not our aim to describe the entire CUF language, for which we refer the reader to the document [DD93] or [Dor94]. Our attention is focused entirely upon the type language, and we are selective rather than encyclopedic in our presentation of details. As our primary goal is to convey the main idea rather than to provide a complete theory, the style is less formal than in the body of the paper; we often express our ideas with annotated examples rather than with rigorously presented algorithms. However, the ideas are quite straightforward, and it should be no problem to fill in the details, if so desired.

We begin with a description of the fragment of the CUF language with which we work.

A.1 The CUF type language We work only with the fragment of CUF that deals with types; we do not address features on types, or the more general recursive sorts. Also, for the moment, we do not deal with constant types. They will be discussed in A.3 below.

The starting point is a set of type symbols $\mathcal{T}$. CUF allows "identifier names," in typical programming language style, but for our purposes, we take $\mathcal{T}$ to be any set. This set may be infinite, but only a finite number of type names are used in any application. From $\mathcal{T}$, we build Boolean expressions from the binary disjunction operator (denoted `;`), the binary conjunction operator (denoted `&`), and the negation operator (denoted `~`). The usual convention of `~` having a higher precedence than either `;` or `&` is followed. To simplify matters, in this discussion, we will assume that `;` and `&` have the same precedence, and that ambiguity must be resolved by the use of parentheses. (CUF actually has rules which disambiguate expressions which use operations which would otherwise have the same precedence.) A typical CUF expression is shown below.

```
t1 & t2 & t3 & (t4 ; ~t5 ; ~(t6 & ~t3) ; t7) & t8
```

An *assignment* is a sentence of the form `t = E`, in which `t` is a type and `E` is an expression. A *statement* is either an expression or an assignment. A statement is always terminated by a period. Thus,

```
t = t1 & t2 & t3 & (t4 ; ~t5 ; ~(t6 & ~t3) ; t7) & t8.
```

is a typical CUF assignment. Think of a CUF *type program* as a set of statements.

In CUF, an expression is thought of as a constraint which must be satisfied. Thus, the constraint `t1 & t2` means that the types `t1` and `t2` cannot have an empty intersection, while the constraint `~(t1 & t2)` means that `t1` and `t2` are disjoint types. The latter constraint would be expressed by $\bigwedge\{\mathtt{t1},\mathtt{t2}\} = \bot$ in a BPPO prespecification; we will discuss the former constraint in A.2 below. An assignment associates an expression with a type. Thus, as expected, `t = t1 & t2` means the same thing as $\bigwedge\{\mathtt{t1},\mathtt{t2}\} = \mathtt{t}$ would in a BPPO prespecification.

CUF admits certain syntactic abbreviations. First of all, an expression of the form `E1 < E2`, with `E1` and `E2` expressions, is an abbreviation for `~(E1) ; E2`. Thus, `t1 < t2` is an abbreviation for `~t1 ; t2`. An expression of the form `t1 | t2 | .. | tn` expresses that the n types are disjoint, and is an abbreviation for the expression

$$(\mathtt{t1\ ;\ t2\ ;\ \ldots\ ;\ tn})\ \&\ (\mathop{\&}_{i<j}(\mathtt{\sim ti\ ;\ \sim tj}))$$

The assignment `t = t1 | t2 | .. | tn` expresses that `t = t1 ; t2 ; .. ; tn` and that the n types are disjoint. Similarly, `t < t1 | t2 | .. | tn` expresses that `t < (t1 ; t2 ; .. ; tn)` and that the n types are disjoint.

Although the overall CUF type system has the types `bot` (corresponding to $\bot$) and `top` (corresponding to $\top$), these types may not be used explicitly in type specifications.

A.2 Translation of CUF expressions to BPPO prespecifications. In the process of translating CUF programs to BPPO prespecifications, it is assumed that there is an unlimited supply of new type symbols, and that such symbols may be generated upon demand. It is further assumed that the CUF program is free of the abbreviation symbols "`|`" and "`<`"; or rather that any expression involving them has been converted to one involving the more primitive Boolean operators, as described in A.1 above.

We will illustrate the translation process as it operates on the example CUF program shown below.

```
t1 & t5 & t9.
t = t1 & t2 & t3 & (t4 ; ~t5 ; ~(t6 & ~t3) ; t7) & t8.
```

The process proceeds as follows. First of all, since a BPPO prespecification does not allow expressions as statements (except for rules of the form $a < b$), for each statement that is an expression `E` (as opposed to an assignment statement), we introduce a new type symbol `s` and replace the expression `E` with `s = E`. In the program above, we replace the expression `t1 & t5 & t9` with an assignment to the new variable `s0`, yielding the following program

```
s0 = t1 & t5 & t9.
t =  t1 & t2 & t3 & (t4 ; ~t5 ; ~(t6 & ~t3) ; t7) & t8.
```

Having translated each statement which is an expression to an assignment, we proceed to simplify the assignment statements to the point at which they may be interpreted as BPPO rules. The approach is as follows. First of all, for each occurrence of negation which occurs in a scope wider than that of a single type symbol, we generate a new type symbol and define the negated component separately. Continuing with the example program introduced above, with the new type symbol `s1` we get

```
s0 = t1 & t5 & t9.
t  = t1 & t2 & t3 & (t4 ; ~t5 ; ~s1 ; t7) & t8.
s1 = t6 & ~t3.
```

Next, for each negated type we define an explicit new type symbol corresponding to its complement. Continuing with this example, we introduce the new type symbols `s2`, `s3`, and `s4`, which represent the complements `~t5`, `s1`, and `t3`, respectively. We also introduce a new symbol `btop`, the "bogus top" symbol, and we have the following declarations.

```
s0 = t1 & t5 & t9.
t  = t1 & t2 & t3 & (t4 ; s2 ; s3 ; t7) & t8.
s1 = t6 & s4.
s2 | t5.
s3 | s1.
s4 | t3.
btop = s2 ; t5.
btop = s3 ; s1.
btop = s4 ; t3.
```

Next, we decompose all right-hand sides of assignment statements which contain nestings of conjunction and disjunction, so that each right-hand side has only one form. In our example, we introduce one more new type symbol s5, and have as our final result the following.

```
s0 = t1 & t5 & t9.
t  = t1 & t2 & t3 & s5 & t8.
s1 = t6 & s4.
s2 | t5.
s3 | s1.
s4 | t3.
btop = s2 ; t5.
btop = s3 ; s1.
btop = s4 ; t3.
s5 = t4 ; s2 ; s3 ; t7.
```

Finally, we need to ensure that btop is the largest type. To so do, we introduce a statement of the form $\mathtt{t} < \mathtt{btop}$ for each type t, either from the regular program or from the set which we added (except for btop itself). Thus, all told, we get the program shown in Figure A.1. We now have a CUF program which corresponds directly to a BPPO prespecification. To get the prespecification, we simply translate in the obvious fashion; it is $\mathbf{S} = (P, \mathbf{C})$ with

$$P = \{\mathtt{t}, \mathtt{t1}, \mathtt{t2}, \mathtt{t3}, \mathtt{t4}, \mathtt{t5}, \mathtt{t6}, \mathtt{t7}, \mathtt{t8}, \mathtt{t9}, \mathtt{s0}, \mathtt{s1}, \mathtt{s2}, \mathtt{s3}, \mathtt{s4}, \mathtt{s5}\},$$

and with $\mathbf{C}$ shown in Figure A.1. Note that we are interpreting btop as the $\top$ in $\mathbf{S}$, so there is no need to explicitly translate the statements of the form `t < btop`, since they are automatically implied in any BPPO.

This is *almost* a BPPO prespecification which is equivalent to the original CUF program. But there are a few points of difference. Most importantly, we have introduced new variables. If we want the BPPO prespecification to be truly equivalent to the CUF program, then we cannot demand that these new variables be separable from each other or from the original variables in the same vein that the original variables might be from each other, since these new variables are not part of the original specification. Thus, we have to be more selective about which variables we demand be separable. To address this issue, we maintain two sets of pairs of types, which we call the *mandatory pairs* and the *regular pairs*. We denote these pairs by $\mathsf{MandPair}$ and $\mathsf{RegPair}$, respectively.

The mandatory pairs are those which must be separated in any distributive extension of the BPPO prespecification which is considered to faithfully model the CUF program. When we translate an expression statement E to an assignment statement `s = E`, by introducing the new variable s, we add the pair $\{\mathtt{s}, \perp\}$ to the collection of mandatory separation pairs. This is because the semantics of the CUF expression statement E is precisely that E must be satisfiable. Hence s cannot be the empty type. Furthermore, for each type t that was used in the original CUF program, we add the pair $\{\mathtt{t}, \perp\}$. This just says that the original types must be distinct from the empty type, which CUF requires in its consistency checks.

The regular pairs are those that must be separated to yield an injective distributive extension. For each type t that was used in the original CUF program, we

```
s0 = t1 & t5 & t9.
t  = t1 & t2 & t3 & s5 & t8.
s1 = t6 & s4.
s2 | t5.
s3 | s1.
s4 | t3.
btop = s2 ; t5.
btop = s3 ; s1.
btop = s4 ; t3.
s5 = t4 ; s2 ; s3 ; t7.
s0 < btop.
s1 < btop.
s2 < btop.
s3 < btop.
s4 < btop.
s5 < btop.
t1 < btop.
t2 < btop.
t3 < btop.
t4 < btop.
t5 < btop.
t6 < btop.
t7 < btop.
t8 < btop.
t9 < btop.
```

$$
\begin{aligned}
\mathbf{C} = \{ \quad & (\bigwedge\{\mathtt{t1},\mathtt{t5},\mathtt{t9}\} = \mathtt{s0}) \\
& (\bigwedge\{\mathtt{t1},\mathtt{t2},\mathtt{t3},\mathtt{s5},\mathtt{t8}\} = \mathtt{t}) \\
& (\bigwedge\{\mathtt{t6},\mathtt{s4}\} = \mathtt{s1}) \\
& (\bigwedge\{\mathtt{s2},\mathtt{s5}\} = \bot) \\
& (\bigwedge\{\mathtt{s3},\mathtt{s1}\} = \bot) \\
& (\bigwedge\{\mathtt{s4},\mathtt{t3}\} = \bot) \\
& (\bigvee\{\mathtt{t6},\mathtt{s4}\} = \mathtt{s1}) \\
& (\bigvee\{\mathtt{s2},\mathtt{s5}\} = \top) \\
& (\bigvee\{\mathtt{s3},\mathtt{s1}\} = \top) \\
& (\bigvee\{\mathtt{s4},\mathtt{t3}\} = \top) \\
& (\bigvee\{\mathtt{t4},\mathtt{s2},\mathtt{s3},\mathtt{t7}\} = \mathtt{s5})\}
\end{aligned}
$$

Figure A.1: The final transformation of the example CUF program, and the corresponding BPPO prespecification.

add the pair and $\{\mathtt{t}, \top\}$. This just says that the original types must be distinct from the universe. Furthermore, for each pair of variables $\{\mathtt{t1}, \mathtt{t2}\}$ from the original

CUF program, we add that pair to the collection of regular pairs. Note specifically that we do not require that types which we introduced in the translation process be separable from each other, from the original types, or even from $\perp$, except in the case that we translated an expression statement to an assignment statement. Thus, in asking for a "legitimate" extension, we ask for Q-separability, with Q a set which contains all mandatory pairs, and whatever subset of regular pairs is deemed appropriate for the particular application.

The second issue that we must address is the bogus top `btop`; we must reconcile how that relates to the "real" top `top`. This is quite simple. Although `btop` may not be the top element in the CUF hierarchy (since CUF allows more complex types, such as list types, which we do not consider here), it will be the top element in the set of constructed types, because we have declared it to be such. Thus, while `~t` which we construct may not have the same denotation in a full CUF program as it has here, this will not affect conditions of separability or extensibility in any way.

A.3 Constant types in CUF In addition to ordinary types, the CUF type language allows *constant types*. They are indicated syntactically by using set brackets. Thus, for example, the following statement declares type `t` to consist of the three constant types `a`, `b`, and `c`.

```
r = {a, b, c}.
```

The semantics of this statement is the same as that of

```
r = a ; b ; c.
```

together with a declaration that the three types on the right hand side of the declaration are atomic. Declaration of atoms may not occur within more complex statements, so that an expression such as

```
t = t1 & (t2 ; {a, b, c}).
```

is not legal in CUF.

Our framework already provides the means to handle constants, as described in Sections 1.5 and 2.5. To handle a CUF statement such as

```
r = {a, b, c}.
```

we simply replace it with

```
r = a ; b ; c.
```

in the CUF program. This means that the constraint $\bigvee\{\mathtt{a}, \mathtt{b}, \mathtt{c}\} = \mathtt{r}$ is added to the set of rules of the corresponding BPPO prespecification, and that is added $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ to the set of elements to be rendered atomic.

We have thus shown that CUF programs may be translated to finite BPPO prespecifications with essentially the same semantics. We now illustrate the translation in the opposite direction.

A.4 Translation from BPPO prespecifications to CUF For the most part, anything that can be represented in a finite BPPO prespecification can be represented in a CUF program. Indeed, a rule of the form $\mathtt{a} < \mathtt{b}$ may be realized as the following CUF statement

```
a < b.
```

The rules $\bigvee\{\mathtt{s1},\ldots,\mathtt{sk}\} = \mathtt{a}$ and $\bigvee\{\mathtt{t1},\ldots,\mathtt{tm}\} = \mathtt{b}$ may be realized by the following two rules.

```
a = s1 ; ... ; sn.
b = t1 & ... & tn.
```

The only difficulty arises when we have rules involving $\perp$ or $\top$, as CUF does not allow them to be used explicitly in rules. Now CUF does allow a direct equivalent of a two-element meet rule, as $\bigwedge\{\mathtt{s1},\mathtt{s2}\} = \perp$ may be realized as

```
s1 | s2.
```

However,

```
s1 | s2 | s3.
```

is not equivalent to $\bigwedge\{\mathtt{s1},\mathtt{s2},\mathtt{s3}\} = \perp$, but rather to the three rules $\bigwedge\{\mathtt{s1},\mathtt{s2}\} = \perp$, $\bigwedge\{\mathtt{s1},\mathtt{s3}\} = \perp$, and $\bigwedge\{\mathtt{s2},\mathtt{s3}\} = \perp$ taken together. The only apparent way to model effectively this sort of BPPO prespecification constraint in CUF is to create an an "artificial $\perp$" `abot` in the prespecification. Thus, we replace every rule of the form $\bigwedge S = \perp$ with the rule $\bigwedge S = \mathtt{abot}$. The corresponding CUF statement becomes

```
abot = s1 & s2 & s3.
```

For each type name `t` in the domain under consideration, if we want every declared type to have a nonempty instantiation (as CUF requires), we must also add a statement of the form

```
t & ~abot
```

to ensure that `t` and `abot` do not have the same instantiation.

Similarly, to model a constraint of the form $\bigvee\{\mathtt{s1},\ldots,\mathtt{sk}\} = \top$, we must introduce an "artificial $\top$," and replace the rule with $\bigvee\{\mathtt{s1},\ldots,\mathtt{sk}\} = \mathtt{atop}$. The associated CUF statement becomes

```
atop = s1 ; s2 ; s3.
```