[99] J. Xu. *A Theory of Types and Type Inference in Logic Programming Languages*. PhD thesis, SUNY at Stony Brook, 1989.

[100] E. Yardeni, T. Fruehwirth, and E. Shapiro. Polymorphically typed logic programs. In *Intl. Conference on Logic Programming*, Paris, France, June 1991.

[101] E. Yardeni and E. Shapiro. A type system for logic programs. In E. Shapiro, editor, *Concurrent Prolog*, volume 2. MIT Press, 1987.

[102] C. Zaniolo. The database language GEM. In *ACM SIGMOD Conference on Management of Data*, pages 423–434, 1983.

[103] C. Zaniolo, H. Aït-Kaci, D. Beech, S. Cammarata, L. Kerschberg, and D. Maier. Object-oriented database and knowledge systems. Technical Report DB-038-85, MCC, 1985.

[83] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *IEEE Symposium on Logic Programming*, pages 140–159, 1988.

[84] K.A. Ross. Relations with relation names as arguments: Algebra and calculus. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 1992.

[85] M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for ¬1NF relational databases. Technical Report 84-36, Univ. of Texas at Austin, 1985.

[86] J.W. Schmidt. Some high-level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, September 1977.

[87] D.W. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, pages 140–173, 1981.

[88] M. Stefik and D.G. Bobrow. Object-oriented programming: Themes and variations. *The AI Magazine*, pages 40–62, January 1986.

[89] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[90] K. Thirunarayan and M. Kifer. A theory of nonmonotonic inheritance based on annotated logic. *Artificial Intelligence*, 60(1):23–50, March 1993.

[91] D.S. Touretzky. *The Mathematics of Inheritance*. Morgan-Kaufmann, Los Altos, CA, 1986.

[92] D.S. Touretzky, J.F. Horty, and R.H. Thomason. A clash of intuitions: The current state of nonmonotonic multiple inheritance systems. In *Intl. Joint Conference on Artificial Intelligence*, pages 476–482, 1987.

[93] J.D. Ullman. Database theory: Past and future. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–10, 1987.

[94] J.F. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1*. Computer Science Press, 1988.

[95] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–10, 1989.

[96] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 221–230, 1988.

[97] P. Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. MIT Press, 1987.

[98] J. Wu. *A Theory of Types and Polymorphism in Logic Programming*. PhD thesis, SUNY at Stony Brook, 1992.

[68] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *Proceedings of OOPSLA-86*, pages 472–482, 1986.

[69] J. McCarthy. First order theories of individual concepts and propositions. In J.E. Hayes and D. Michie, editors, *Machine Inteligence*, volume 9, pages 129–147. Edinburgh University Press, 1979.

[70] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[71] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[72] M. Minsky. A framework for representing knowledge. In J. Haugeland, editor, *Mind design*, pages 95–128. MIT Press, Cambridge, MA, 1981.

[73] P. Mishra. Towards a theory of types in Prolog. In *IEEE Symposium on Logic Programming*, pages 289–298, 1984.

[74] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *ACM Symposium on Principles of Programming Languages*, pages 109–124, 1990.

[75] K. Morris, J. Naughton, Y. Saraiya, J. Ullman, and A. Van Gelder. YAWN! (Yet another window on NAIL!). *IEEE Database Engineering*, 6:211–226, 1987.

[76] A. Motro. BAROQUE: A browser for relational databases. *ACM Trans. on Office Information Systems*, 4(2):164–181, 1986.

[77] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, 1989.

[78] E. Neuhold and M. Stonebraker. Future directions in DBMS research (The Laguna Beech report). *SIGMOD Record*, 18(1), March 1989.

[79] G. Phipps, M.A. Derr, and K.A. Ross. Glue-Nail: A deductive database system. In *ACM SIGMOD Conference on Management of Data*, pages 308–317, 1991.

[80] H. Przymusinska and M. Gelfond. Inheritance hierarchies and autoepistemic logic. In *Intl. Symposium on Methodologies for Intelligent Systems*, 1989.

[81] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[82] T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 11–21, 1989.

[52] M. Kifer and J. Wu. A logic for programming with complex objects. *Journal of Computer and System Sciences*, 46, 1993. A special issue from PODS-89. To appear.

[53] W. Kim, J. Banerjee, H-T. Chou, J.F. Garza, and D. Woelk. Composite object support in an object-oriented database system. In *Proceedings of OOPSLA-87*, 1987.

[54] P.G. Kolaitis and C.H. Papadimitriou. Why not negation by fixpoint. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 231–239, 1988.

[55] R. Krishnamurthy and S. Naqvi. Towards a real horn clause language. In *Intl. Conference on Very Large Data Bases*, 1988.

[56] T. Krishnaprasad, M. Kifer, and D.S. Warren. On the circumscriptive semantics of inheritance networks. In *Intl. Symposium on Methodologies for Intelligent Systems*, pages 448–457, 1989.

[57] T. Krishnaprasad, M. Kifer, and D.S. Warren. On the declarative semantics of inheritance networks. In *Intl. Joint Conference on Artificial Intelligence*, pages 1099–1103, 1989.

[58] G. Kuper and M.Y. Vardi. A new approach to database logic. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1984.

[59] G.M. Kuper. An extension of LPS to arbitrary sets. Technical report, IBM, Yorktown Heights, 1987.

[60] G.M. Kuper. Logic programming with sets. *Journal of Computer and System Sciences*, 41(1):44–64, August 1990.

[61] E. Laenens, D. Sacca, and D. Vermeir. Extending logic programming. In *ACM SIGMOD Conference on Management of Data*, pages 184–193, June 1990.

[62] E. Laenens and D. Vermeir. A fixpoint semantics for ordered logic. *Journal Logic and Computation*, 1(2):159–185, 1990.

[63] C. Lecluse and P. Richard. The O_2 database programming language. In *Intl. Conference on Very Large Data Bases*, August 1989.

[64] J.W. Lloyd. *Foundations of Logic Programming (Second Edition)*. Springer Verlag, 1987.

[65] D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Databases and Logic Programming*, pages 6–26, Washington D.C., August 1986.

[66] D. Maier. Why database languages are a bad idea (position paper). In *Proc. of the Workshop on Database Programming Languages*, Roscoff, France, September 1987.

[67] D. Maier. Why isn't there an object-oriented data model. Technical report, Oregon Graduate Center, May 1989.

[38] H. Geffner and T. Verma. Inheritance = chaining + defeat. In *Intl. Symposium on Methodologies for Intelligent Systems*, pages 411–418, 1989.

[39] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium*, pages 1070–1080, 1988.

[40] R.P. Hall. Computational approaches to analogical reasoning: A comparative study. *Artificial Intelligence*, 30:39–120, 1989.

[41] P.J. Hayes. The logic for frames. In D. Metzing, editor, *Frame Conception and Text Understanding*, pages 46–61. Walter de Gruyter and Co., 1979.

[42] P. Hill and R. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. The MIT Press, 1992.

[43] J.F. Horty, R.H. Thomason, and D.S. Touretzky. A skeptical theory of inheritance in nonmonotonic semantic nets. In *National Conference on Artificial Intelligence*, pages 358–363, 1987.

[44] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Intl. Conference on Very Large Data Bases*, pages 455–468, Brisbane, Australia, 1990.

[45] F.N. Kesim and M. Sergot. On the evolution of objects in a logic programming framework. In *Proceedings of the Intl. Conference on Fifth Generation Computer Systems*, pages 1052–1060, Tokyo, Japan, June 1992.

[46] S.N. Khoshafian and G.P. Copeland. Object identity. In *Proceedings of OOPSLA-86*, pages 406–416, 1986.

[47] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *ACM SIGMOD Conference on Management of Data*, pages 393–402, June 1992.

[48] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance and schema. In *ACM SIGMOD Conference on Management of Data*, pages 134–146, 1989.

[49] M. Kifer and G. Lausen. Behavioral inheritance in F-logic. in preparation, 1993.

[50] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 379–393, March 1989.

[51] M. Kifer and J. Wu. A first-order theory of types and polymorphism in logic programming. In *Intl. Symposium on Logic in Computer Science*, pages 310–321, Amsterdam, The Netherlands, July 1991. Expanded version: TR 90/23 under the same title, Department of Computer Science, University at Stony Brook, July 1990.

[22] G. Brewka. The logic of inheritance in frame systems. In *Intl. Joint Conference on Artificial Intelligence*, pages 483–488, 1987.

[23] P. Buneman and R.E. Frankel. FQL - A functional query language. In *ACM SIGMOD Conference on Management of Data*, pages 52–58, 1979.

[24] P. Buneman and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, 1989.

[25] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2):138–164, February 1988.

[26] M. Carey, D. DeWitt, and S. Vanderberg. A data model and query language for EXODUS. In *ACM SIGMOD Conference on Management of Data*, 1988.

[27] C.L. Chang and R.C.T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[28] W. Chen. A theory of modules based on second-order logic. In *IEEE Symposium on Logic Programming*, pages 24–33, September 1987.

[29] W. Chen and M. Kifer. Sorts, types and polymorphism in higher-order logic programming. Technical Report 92-CSE-7, Department of Computer Science and Engineering, Southern Methodist University, March 1992.

[30] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[31] W. Chen and D.S. Warren. C-logic for complex objects. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 369–378, March 1989.

[32] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer Verlag, 1981.

[33] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. In *ACM Symposium on Principles of Programming Languages*, pages 125–136, 1990.

[34] G. Dobbie and R. Topor. A model for inheritance and overriding in deductive object-oriented systems. In *Sixteenth Australian Computer Science Conference*, January 1993.

[35] H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[36] D.W. Etherington and R. Reiter. On inheritance hierarchies with exceptions. In *National Conference on Artificial Intelligence*, pages 104–108, Washington, D.C., 1983.

[37] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of ACM*, 28(9):904–920, 1985.

[8] R. Anderson and W.W. Bledsoe. A linear format resolution with merging and a new technique for establishing completeness. *Journal of ACM*, 17(3):525–534, 1970.

[9] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Intl. Conference on Deductive and Object-Oriented Databases*, pages 40–57, 1989.

[10] M. Balaban. The generalized concept formalism – A frames and logic based representation model. In *Proceedings of the Canadian Artificial Intelligence Conference*, pages 215–219, 1986.

[11] M. Balaban and S. Strack. LOGSTER — A relational, object-oriented system for knowledge representation. In *Intl. Symposium on Methodologies for Intelligent Systems*, pages 210–219, 1988.

[12] F. Bancilhon. Object-oriented database systems. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 152–162, 1988.

[13] F. Bancilhon and S.N. Khoshafian. A calculus of complex objects. *Journal of Computer and System Sciences*, 38(2):326–340, April 1989.

[14] J. Banerjee, W. Kim, and K.C. Kim. Queries in object-oriented databases. In *Proc. of the 4-th Intl. Conf. on Data Engineering*, Los Angeles, CA, February 1988.

[15] C. Beeri. Formal models for object-oriented databases. In *Intl. Conference on Deductive and Object-Oriented Databases*, pages 370–395. Elsevier Science Publ., 1989.

[16] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (LDL). Technical report, MCC, 1987.

[17] C. Beeri, R. Nasr, and S. Tsur. Embedding ψ-terms in a horn-clause logic language. In *Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 347–359. Morgan Kaufmann, 1988.

[18] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–300, April 1991.

[19] A.J. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-270, University of Toronto, April 1992. Revised: April 1993. Available in csri-technical-reports/270/report.ps by anonymous ftp to *csri.toronto.edu*.

[20] A.J. Bonner and M. Kifer. Transaction logic programming. In *Intl. Conference on Logic Programming*, Budapest, Hungary, June 1993. To appear.

[21] S. Brass and U.W. Lipeck. Semantics of inheritance in logical object specifications. In *Intl. Conference on Deductive and Object-Oriented Databases*, pages 411–430, December 1991.

1. *Unification of sets:*
   This is not a problem specific to F-logic but, rather, is a fact of life. Every language that allows the use of sets in any essential way has to put up with the exponential worst-case complexity of set-unification.

2. *Permutation of methods inside molecules:*
   Since methods may be denoted by *nonground* id-terms, they can match each other in many different ways. For instance, in unifying $P[X \rightarrow V;\ Y \rightarrow W]$ and $P[X' \rightarrow V';\ Y' \rightarrow W']$, the method denoted by X can match either X' or Y'; similarly for Y. Thus, an extra complexity is expected due to the higher-order syntax of F-logic.

The key factor in estimating the complexity of unification in "practical" cases is the number of atoms comprising each molecule in the bodies of the rules. For, if n_1 is the number of atoms in T_1 and n_2 is the same for T_2, then the number of unifiers of T_1 into T_2 is bounded by $n_2^{n_1}$. Now, to resolve a pair of rules $\ldots \longleftarrow \ldots, T_1, \ldots$ and $T_2 \longleftarrow \ldots,$ where T_1 and T_2 are data or signature molecules, we need to unify T_1 into T_2 and, therefore, the above parameter, n_1, which denotes the number of atoms in T_1, is most crucial. However, our experience with writing F-programs and all the examples in this paper show that this number is usually very small (≤ 2) and so, we believe, combinatorial explosion is unlikely to happen in practice.

References

[1] S. Abiteboul and C. Beeri. On the power of languages for manipulation of complex objects. Technical report, Rapport de Recherche INRIA, 1988. To appear in TODS.

[2] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. In *Workshop on Database Programming Languages*, pages 253–276, Roscoff, France, September 1987.

[3] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *ACM SIGMOD Conference on Management of Data*, pages 159–173, 1989.

[4] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 240–250, 1988.

[5] H. Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351, 1986.

[6] H. Aït-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[7] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. In *Programming Languages Implementation and Logic Programming (PLILP'91)*, pages 255–274, Passau, Germany, August 1991.

Input : Pair of molecules, T_1 and T_2.
Output : A complete set Ω of mgu's of T_1 *into* T_2.

1. If $id(T_1)$ and $id(T_2)$ are unifiable, then set $\theta := UNIFY(< id(T_1) >, < id(T_2) >)$.
   Otherwise, stop: T_1 and T_2 are not unifiable.

2. If T_1 is of the form $S[\,]$ (a degenerated molecule), then stop: θ is the only mgu.

3. Set $\Omega := \{\}$.
   for each mapping $\lambda \in Maps(T_1, T_2)$ **do**:
   Set $\sigma_\lambda := \theta$.
   for each atom φ in $atoms(T_1)$ **do**:
   Let $\psi \stackrel{\text{def}}{=} \lambda(\varphi)$.
   if $val(\varphi) = \emptyset$ **then** Unify the tuples $\sigma_\lambda(\vec{S_1})$ and $\sigma_\lambda(\vec{S_2})$, where
   $\vec{S_1} = < method(\varphi), arg_1(\varphi), ..., arg_n(\varphi) >$ and
   $\vec{S_2} = < method(\psi), arg_1(\psi), ..., arg_n(\psi) >$.
   if $\sigma_\lambda(\vec{S_1})$ and $\sigma_\lambda(\vec{S_2})$ are unifiable **then** set $\sigma_\lambda := UNIFY(\sigma_\lambda(\vec{S_1}), \sigma_\lambda(\vec{S_2})) \circ \sigma_\lambda$.
   else discard this σ_λ and jump out of the inner loop to select another λ.
   fi
   else
   Unify tuples $\sigma_\lambda(\vec{S_1})$ and $\sigma_\lambda(\vec{S_2})$, where
   $\vec{S_1} = < method(\varphi), arg_1(\varphi), ..., arg_n(\varphi), val(\varphi) >$ and
   $\vec{S_2} = < method(\psi), arg_1(\psi), ..., arg_n(\psi), val(\psi) >$.
   If $\sigma_\lambda(\vec{S_1})$ and $\sigma_\lambda(\vec{S_2})$ are unifiable **then** set $\sigma_\lambda := UNIFY(\sigma_\lambda(\vec{S_1}), \sigma_\lambda(\vec{S_2})) \circ \sigma_\lambda$.
   else discard this σ_λ and jump out of the inner loop to select another λ.
   fi
   fi
   Set $\Omega := \Omega \cup \{\sigma_\lambda\}$.
   od
   od

4. Return Ω, a complete set of mgu's of T_1 into T_2.

Figure 10: Computing a Complete Set of MGU's

Lemma B.1 *The algorithm in Figure 10 correctly finds a complete set of mgu's of* T_1 *into* T_2.

Proof: Clearly, all elements of Ω are mgu's of T_1 into T_2, so we will only show that Ω is complete. Consider a unifier θ of T_1 into T_2. By definition, there is a mapping $\lambda \in Maps(T_1, T_2)$ from $atoms(T_1)$ to $atoms(T_2)$, such that θ maps every φ in $atoms(T_1)$ into an appropriate atom $\lambda(\varphi)$ in $atoms(T_2)$. Clearly, substitution σ_λ constructed in the inner for-loop in Step 3 of the above algorithm is a most general unifier that maps every $\varphi \in atoms(T_1)$ to $\lambda(\varphi) \in atoms(T_2)$. So, θ is an instance of σ_λ.

Summarizing, we have shown that 1) each element of Ω is an mgu; and 2) if θ is a unifier of T_1 into T_2, then θ is an instance of an element $\sigma_\lambda \in \Omega$. Hence, Ω is a complete set of mgu's of T_1 into T_2. □

In the worst-case, the algorithm of Figure 10 may yield an exponential number of unifiers, which may happen due to the following two factors:

Proposition A.6 *Every locally stratified F-program has a unique perfect H-model.*

Proof: (Sketch) The proof is similar to that in [81]. Another, indirect proof can be derived from Theorem 17.1. Let $\Gamma(\mathbf{P}^a)$ be a translation of $\mathbf{P}^a$ into classical logic. To prove the result, it is necessary to show that if $\prec_{\mathbf{P}^a}$ is well-founded then so is $\prec_{\Gamma(\mathbf{P}^a)}$, where the latter is the order on ground atoms that is used in the usual definition of local stratification. Then, it must be shown that (in the notation of Theorem 17.1) Φ maps $\ll_{\mathbf{P}^a}$-minimal models of $\mathbf{P}$ into $\ll_{\Gamma(\mathbf{P}^a)}$-minimal models of $\Gamma(\mathbf{P}^a)$. □

B Appendix: A Unification Algorithm for F-molecules

This appendix presents an algorithm for finding a complete set of mgu's for a pair of molecules. We remind (Section 10.1) that all complete sets of mgu's are equivalent to each other and therefore it suffices to find just one such set. Recall that a molecule can be represented as a conjunction of its constituent atoms. For a pair of molecules, T_1 and T_2, an mgu of T_1 *into* T_2 is a *most general* substitution that turns every constituent atom of T_1 into a constituent atom of T_2. Different correspondences between the constituent atoms of T_1 and T_2 may lead to different mgu's. To find a *complete* set of mgu's, we must take all such correspondences into account. Let φ be an atom of either of the following forms:

(i) $P[Method @ Q_1, ..., Q_k \rightsquigarrow R]$, where $\rightsquigarrow$ denotes any of the six allowed types of arrows in method expressions;

(ii) $P[Method @ Q_1, ..., Q_k \rightsquigarrow \{\,\}]$, where $\rightsquigarrow$ denotes $\rightarrow\!\!\!\rightarrow$ or $\bullet\!\!\rightarrow\!\!\!\rightarrow$; or

(iii) $P[Method @ Q_1, ..., Q_k \rightsquigarrow (\,)]$, where $\rightsquigarrow$ denotes $\Rightarrow$ or $\Rightarrow\!\!\!\!\!\Rightarrow$.

The following notation is used in the unification algorithm in Figure 10:

$$
\begin{aligned}
id(\varphi) &= P, \\
method(\varphi) &= Method, \\
arg_i(\varphi) &= Q_i, \text{ for } i = 1...k, \\
val(\varphi) &= \begin{cases} R & \text{if } \varphi \text{ is of the form (i) above} \\ \emptyset & \text{if } \varphi \text{ has the form (ii) or (iii) above} \end{cases}
\end{aligned}
$$

In addition, if T is a molecule then $atoms(T)$ will denote the set of atoms of T. If T_1 and T_2 are molecules then $Maps(T_1, T_2)$ denotes the collection of mappings $\{\lambda : atoms(T_1) \longmapsto atoms(T_2)\}$ that preserves method arities and the type of the method expression (or, in other words, the type of the arrow used in those expressions).

As noted in Section 10.1, the mgu of a pair of tuples of id-terms, $\langle P_1, ..., P_n\rangle$ and $\langle Q_1, ..., Q_n\rangle$, coincides with the mgu of the terms $f(P_1, ..., P_n)$ and $f(Q_1, ..., Q_n)$, where f is an arbitrary n-ary function symbol. So, we can use any standard unification procedure to unify tuples of id-terms. Given a pair of tuples of id-terms $\vec{S_1}$ and $\vec{S_2}$, we use $UNIFY(\vec{S_1}, \vec{S_2})$ to denote a procedure that returns the mgu of $\vec{S_1}$ and $\vec{S_2}$, if one exists.

Axioms of the IS-A hierarchy.

$$\begin{array}{rll} X :: Y & \longleftarrow X :: Z \wedge Z :: Y & \% \text{ Transitivity} \\ X \doteq Y & \longleftarrow X :: Y \wedge Y :: X & \% \text{ Acyclicity} \\ X : Y & \longleftarrow X : Z \wedge Z :: Y & \% \text{ Subclass inclusion} \end{array}$$

Axioms of typing. For every arity, n, and for $\approx\!\!>$ standing for either $\Rightarrow$ or $\Rightarrow\!\!>$:

$$\begin{array}{rll} X[M@A_1, ..., A_n \approx\!\!> T] & \longleftarrow & X'[M@A_1, ..., A_n \approx\!\!> T] \wedge X :: X' \\ & & \% \text{ Type inheritance} \\ X[M@A_1, ...A_i, ..., A_n \approx\!\!> T] & \longleftarrow & X[M@A_1, ..., A'_i, ..., A_n \approx\!\!> T] \wedge A_i :: A'_i \\ & & \% \text{ Input restriction} \\ X[M@A_1, ..., A_n \approx\!\!> T] & \longleftarrow & X[M@A_1, ..., A_n \approx\!\!> T'] \wedge T' :: T \\ & & \% \text{ Output relaxation} \end{array}$$

Axioms of scalarity. For every arity, n, and for $\rightsquigarrow$ standing for $\rightarrow$ or $\bullet\!\!\rightarrow$:

$$(V \doteq V') \longleftarrow X[M@A_1, ..., A_n \rightsquigarrow V] \wedge X[M@A_1, ..., A_n \rightsquigarrow V']$$

Definition A.3 (Program Augmentation) For any program, $\mathbf{P}$, define its *augmentation*, $\mathbf{P}^a$, to be the union of $\mathbf{P}$ with the *relevant* ground instances of the closure axioms.

A ground instance of an axiom is *relevant* if all atoms in its premises are relevant. An atom is relevant if and only if it occurs in the head of a rule in $\mathbf{P}^*$ or, recursively, in the head of a axiom-instance whose relevance was established previously. □

We can now rectify the earlier definitions by considering the order $\prec_{\mathbf{P}^a}$ instead of $\prec_{\mathbf{P}}$, *i.e.*, by using augmentations of programs rather than programs themselves.

Definition A.4 (Locally Stratified Programs) An F-program $\mathbf{P}$ is *locally stratified* if the relation "$\prec_{\mathbf{P}^a}$" is *well-founded*. □

Definition A.5 (Perfect Models) H-models of $\mathbf{P}$ that are minimal with respect to $\ll_{\mathbf{P}^a}$ are called *perfect* models of $\mathbf{P}$. □

Returning to programs $\mathbf{P}_1$ and $\mathbf{P}_2$ in (29), we can now see that they are not locally stratified with respect to Definition A.4 (while they were locally stratified with respect to Definition A.1). Indeed, consider the following instance of the paramodulation axiom:

$$p(b) \longleftarrow p(a) \wedge (a \doteq b)$$

Here, both atoms in the premise are relevant. Therefore, we have a negative cycle from $\neg p(b)$ through $p(a)$, and back to $p(b)$ in $\mathbf{D}_{\mathcal{L}}(\mathbf{P}_1^a)$; and from $\neg p(b)$ through $a \doteq b$, and back to $p(b)$ in $\mathbf{D}_{\mathcal{L}}(\mathbf{P}_2^a)$.

Similarly, we can show that (31) has a negative cycle. Consider the following instance of the type inheritance axiom:

$$b[attr \Rightarrow c] \longleftarrow a[attr \Rightarrow c] \wedge b :: a$$

This rule belongs to the augmentation of $\mathbf{P}$ because both of its premises are relevant. Thus, $\mathbf{D}_{\mathcal{L}}(\mathbf{P}_3^a)$ has the following negative cycle: $\neg b[attr \Rightarrow c]$, to $p(a)$, to $a[attr \Rightarrow c]$, and back to $b[attr \Rightarrow c]$.

The unique perfect model of $\mathbf{P}_1$ is $M_1 = \{p(a), p(b), a \doteq b\}$, and $M_2 = \{p(a), p(b)\}$ is the unique perfect model of $\mathbf{P}_2$. (These models are obtained from the definitions in [81] in a simple-minded way, *i.e.*, by considering equality as an ordinary predicate.) The trouble is that $p(b)$ is not supported by any rule. In fact, in both programs, $p(b)$ was derived assuming $\neg p(b)$ is true.

As the above example shows, equality is a problem not only in F-logic but also in classical logic programming. However, in the classical case, this problem is side-stepped by disallowing equality to occur in the rule heads. Unfortunately, this shortcut does not work for F-logic since equality may be derived even if it is not mentioned explicitly. For instance, $\{a :: b, b :: a\} \models (a \doteq b)$ and $\{a[attr \rightarrow b], a[attr \rightarrow c]\} \models (b \doteq c)$.

Another difficulty comes from the closure properties inherent in F-programs. For instance, the built-in property of signature inheritance corresponds to the following deductive rule:

$$X[M @ \vec{args} \Rightarrow Y] \longleftarrow X :: Z \wedge Z[M @ \vec{args} \Rightarrow Y] \tag{30}$$

This rule creates an effect similar to recursive cycles through negation when signatures are negatively dependent on other data. To illustrate the problem, consider the following program:

$$\begin{array}{l} b :: a \\ p(a) \longleftarrow \neg b[attr \Rightarrow c] \\ a[attr \Rightarrow c] \longleftarrow p(a) \end{array} \tag{31}$$

Together with (30), we obtain a negative cycle going from $\neg b[attr \Rightarrow c]$ to $p(a)$ to $a[attr \Rightarrow c]$ to $b[attr \Rightarrow c]$.

Fortunately, there is a simple way out of these difficulties (which also works for classical logic programs with equality). The idea is to account for the idiosyncrasies of the equality and of the built-in features of F-logic by adding additional axioms to $\mathbf{P}$ in the form of logical rules. Each of these new axioms, called *closure* axioms, corresponds to an inference rule of Section 10. The only inference rules that do not need axioms are the rules of resolution, factorization, merging, and elimination.

Axioms of paramodulation. For every arity, n, every predicate symbol, p (including $\doteq$), and for "$\rightsquigarrow$" standing for either of the six arrow types:

$$\begin{array}{rcl} p(Y_1, ..., Y_i', ..., Y_n) & \longleftarrow & p(Y_1, ..., Y_i, ..., Y_n) \wedge (Y_i \doteq Y_i') \\ X[M' @ A_1, ..., A_n \rightsquigarrow V] & \longleftarrow & X[M @ A_1, ..., A_n \rightsquigarrow V] \wedge (M \doteq M') \end{array}$$

There are also rules similar to the last one for the paramodulation on X, V, and on each of the A_i. Notice that paramodulation also accounts for the transitivity and the symmetry of the equality predicate. Additionally, to account for the paramodulation into subterms, the following axiom is needed:

$$(f(X_1, ..., X_i', ..., X_n) \doteq f(X_1, ..., X_i, ..., X_n)) \longleftarrow (X_i \doteq X_i')$$

for every function symbol, $f \in \mathcal{F}$.

"$\prec_{\mathbf{P}}$" and "$\preceq_{\mathbf{P}}$" are not partial orders. However, they are partial orders when $\mathcal{D}_{\mathcal{L}}(\mathbf{P})$ has no cycles that contain negative edges (the *negative* cycles).

Definition A.1 (Locally Stratified Programs, preliminary) An F-program, $\mathbf{P}$, is *locally stratified* if the relation "$\prec_{\mathbf{P}}$" is *well-founded*, *i.e.*, there is no infinite decreasing chain of the form $\cdots \prec_{\mathbf{P}} \phi_2 \prec_{\mathbf{P}} \phi_1$. □

This definition implies that for locally stratified programs "$\prec_{\mathbf{P}}$" is irreflexive and asymmetric (for, otherwise, there would be an infinite chain $\cdots \prec_{\mathbf{P}} \phi_1 \prec_{\mathbf{P}} \phi_2 \prec_{\mathbf{P}} \phi_1$, for some ϕ_1 and ϕ_2 such that $\phi_1 \prec_{\mathbf{P}} \phi_2$ and $\phi_2 \prec_{\mathbf{P}} \phi_1$). Similarly, Definition A.1 implies that the graph $\mathcal{D}_{\mathcal{L}}(\mathbf{P})$ does not have negative cycles when $\mathbf{P}$ is locally stratified.

Next we introduce a *preference* quasi-order as follows: an H-structure $\mathbf{M}$ is *preferable* to $\mathbf{L}$, denoted $\mathbf{M} \ll_{\mathbf{P}} \mathbf{L}$, if whenever there is an atom φ such that $\varphi \in \mathbf{M} - \mathbf{L}$ and then there is an atom ψ such that $\mathtt{level}_{\mathbf{P}}(\psi) < \mathtt{level}_{\mathbf{P}}(\varphi)$, $\psi \in \mathbf{L} - \mathbf{M}$.

Definition A.2 (Perfect Models, preliminary) Let $\mathbf{P}$ be a locally stratified F-program and $\ll_{\mathbf{P}}$ be an associated quasi-order on its H-models. H-models of $\mathbf{P}$ that are minimal with respect to $\ll_{\mathbf{P}}$ are called *perfect* models of $\mathbf{P}$. □

In general, $\ll_{\mathbf{P}}$ is a quasi-order. However, as in [81], it can be shown that for any locally stratified program, $\mathbf{P}$, $\ll_{\mathbf{P}}$ is a partial order on the H-models of $\mathbf{P}$. Also, as in the classical theory, it can be shown that every locally stratified program has a unique perfect model.

Let $\mathbf{P}$ be a locally stratified program. It is easy to verify that there is a function

$$\mathtt{level}_{\mathbf{P}} : \mathcal{HB}(\mathcal{F}) \longmapsto \mathcal{N}$$

where $\mathcal{N}$ is the set of all natural numbers, such that for any pair, $\phi, \psi \in \mathcal{HB}(\mathcal{F})$, if $\phi \preceq_{\mathbf{P}} \psi$ then $\mathtt{level}_{\mathbf{P}}(\phi) \leq \mathtt{level}_{\mathbf{P}}(\psi)$ and if $\phi \prec_{\mathbf{P}} \psi$ then $\mathtt{level}_{\mathbf{P}}(\phi) < \mathtt{level}_{\mathbf{P}}(\psi)$.

Given a level-function, we can partition the rules in $\mathbf{P}^*$ into $\mathbf{P}^*_1$, $\mathbf{P}^*_2$, ... according to the level of the head-literal in each rule. More precisely, each $\mathbf{P}^*_k$ consists of all those rules in $\mathbf{P}^*$ whose head-literal is assigned the level $\leq k$. In particular, $\mathbf{P}^*_k - \mathbf{P}^*_{k-1}$ consists of all rules with head-literals at level k.

As in the classical theory, it can be shown that the perfect model of $\mathbf{P}$ can be computed by firing logical rules in the stratification order determined by $\mathtt{level}_{\mathbf{P}}$, and that the result does not depend on the choice of the level-function as long as it is synchronized with $\prec_{\mathbf{P}}$ and $\preceq_{\mathbf{P}}$.

Unfortunately, the above straightforward adaptation of the notion of perfect models is inadequate. One problem comes from the equality predicate, "$\doteq$". In [81], perfect models were introduced assuming the so called *freeness* axioms for equality, which postulate that ground terms are equal if and only if they are identical. Without these axioms, perfect models do not provide natural semantics even for simple locally stratified programs. For instance, consider the following pair of F-programs:

$$\mathbf{P}_1: \quad \begin{array}{l} a \doteq b \\ p(a) \longleftarrow \neg p(b) \end{array} \qquad\qquad \mathbf{P}_2: \quad \begin{array}{l} a \doteq b \longleftarrow \neg p(b) \\ p(a) \end{array} \tag{29}$$

dimension: F-logic is capable of representing almost all aspects of what is known as the object-oriented paradigm. We have provided a formal semantics for the logic and have shown that it naturally embodies the notions of complex objects, inheritance, methods, and types. F-logic has a sound and complete resolution-based proof procedure, which makes it also computationally attractive and renders it a suitable basis for developing a theory of object-oriented logic programming.

Acknowledgements: We are indebted to Catriel Beeri for the very detailed comments, almost as long as this paper, which helped us improve the presentation. Many other improvements in this paper are due to the suggestions of Rodney Topor. We are grateful to Alexandre Lefebvre and Heinz Uphoff for pointing out a problem with an earlier version of the semantics of inheritance. We also wish to thank Serge Abiteboul, Mira Balaban, Weidong Chen, Rick Hull, David Maier, Ken Ross, Shuky Sagiv, Moshe Vardi, David Warren and Gio Wiederhold for the discussions that helped shape this paper. Last, but not least, thanks goes to the anonymous referees for their constructive critique.

A Appendix: A Perfect-Model Semantics for F-logic

The general principle in adapting the various semantics for negation (such as [81, 39, 96, 95, 82, 4, 54]) to F-logic is to use method names whenever predicates are used in the standard setting. For instance, for the perfect-model semantics [81], stratification must be ensured with respect to method names, and so the program

$$X[wants \twoheadrightarrow Y] \longleftarrow \neg X[has \twoheadrightarrow Y]$$

would be considered as stratified despite the recursion through negation within the same object, X. Note that $\neg x[has \twoheadrightarrow y]$ here means that $y \notin x.has$; it does *not* mean that the value of *has* on x is empty or undefined. In contrast, consider the following one-rule program (adapted from [96]):

$$\begin{aligned} sillyGame[winningPos \twoheadrightarrow Pos_1] \longleftarrow \quad & \neg sillyGame[winningPos \twoheadrightarrow Pos_2] \\ & \wedge\, Pos_1[legalMoves \twoheadrightarrow Pos_2] \end{aligned}$$

It is not considered as locally stratified in F-logic since there is recursion through negation in the method *winningPos* when both Pos_1 and Pos_2 are instantiated to the same constant, say *badPos*.

However, the process of adapting perfect models to F-logic is more involved. We present a solution in two steps. First, we apply the above general principle directly and come up with a preliminary definition. Then we point to the inadequacies of this definition and show how problems can be corrected.

Let $\mathcal{L}$ be an F-logic language with $\mathcal{F}$ being its set of function symbols. Consider a general F-program, $\mathbf{P}$. We define a *dependency graph*, $\mathcal{D}_\mathcal{L}(\mathbf{P})$, as follows: Let $\mathbf{P}^*$ denote the set of all ground instances of $\mathbf{P}$ in $\mathcal{L}$. The *nodes* of $\mathcal{D}_\mathcal{L}(\mathbf{P})$ correspond to the ground atoms in the Herbrand base, $\mathcal{HB}(\mathcal{F})$. A *positive arc*, $\varphi \xleftarrow{\oplus} \psi$, connects a pair of nodes, φ and ψ, if and only if there is a rule, $\bar{\varphi} \longleftarrow \cdots \wedge \bar{\psi} \wedge \cdots$, in $\mathbf{P}^*$ such that φ and ψ are constituent atoms of $\bar{\varphi}$ and $\bar{\psi}$, respectively. A *negative arc*, $\varphi \xleftarrow{\ominus} \psi$, is in $\mathcal{D}_\mathcal{L}(\mathbf{P})$ if and only if $\mathbf{P}^*$ has a rule $\bar{\varphi} \longleftarrow \cdots \wedge \neg\bar{\psi} \wedge \cdots$.

We shall write $\phi \preceq_\mathbf{P} \psi$, where ϕ and ψ are atoms in $\mathcal{HB}(\mathcal{F})$, if $\mathcal{D}_\mathcal{L}(\mathbf{P})$ has a directed path from ϕ to ψ. We write $\phi \prec_\mathbf{P} \psi$ if there is a negative such path (*i.e.*, some arcs on the path are negative). In general,

The semantic mapping Φ can be defined along similar lines. Once the mappings , and Φ are defined in this way, verifying (28) becomes a simple, albeit lengthy, task. Examples of this kind of proofs can be found in [98, 30]. □

The reader who might be puzzled by the revelation that F-logic is, in a sense, equivalent to predicate calculus, may find comfort in the following arguments. First, semantics-by-encoding, as in Theorem 17.1, is inadequate as it is indirect and gives little help to the user when it comes to understanding the meaning of a program. Since our declared goal was to provide a logical rendition for a class of languages that are collectively classified as object-oriented, taking the mapping , of Theorem 17.1 for the meaning of F-logic would be a misnomer. Indeed, , is merely an algorithm that gives little insight into the nature of object-oriented concepts the logic is designed to model, and the proof sketch of Theorem 17.1 should make it clear that even for simple F-programs their , -image is not easily understandable. Another argument—well articulated in the concluding section to [30]—can be summarized as follows: The syntax of a programming language is of paramount importance, as it shapes the way programmers approach and solve problems. Third, a direct semantics for a logic shows a way of defining a proof theory that is tailored to that logic. Such a proof theory is likely to be a better basis for an implementation than the general-purpose proof theory of the classical logic. Lastly, Theorem 17.1 relates only the monotonic part of F-logic to classical logic. Mapping the non-monotonic components of F-logic into a non-monotonic theory for predicate calculus does not lead to a bonified logic.

18 Conclusion

Unlike the relational approach that was based on theoretical grounds from the very beginning, the object-oriented approach to databases was dominated by "grass-roots" activity where several implementations existed (*e.g.*, [53, 68]) without the accompanying theoretical progress. As a result, many researchers had felt that the whole area of object-oriented databases is misguided, lacking direction and needing a spokesman, like Codd, who could "*coerce the researchers in this area into using common set of terms and defining a common goal that they are hoping to achieve* [78]."

Our contention is that the problem lies much deeper: when Codd made his influential proposal, he was relying on a large body of knowledge in Mathematical Logic concerning predicate calculus. He had an insight to see a very practical twist to a rather theoretical body of knowledge about mathematical logic, which has led him to develop a theory that revolutionized the entire database field. Until now, logical foundations for object-oriented databases that are parallel to those underlying the relational theory were lacking and this was a major factor for the uneasy feeling. In his pioneering work [65], Maier proposed a framework for defining a model-theoretic semantics for a logic with an object-oriented syntax. However, he encountered many difficulties with his approach and subsequently abandoned this promising direction. As it turned out, the difficulties were not fatal, and the theory was repaired and significantly extended in [31, 50].

In this paper, we have proposed a novel logic that takes the works reported in [31, 50] to a new

of canonic models.

Because of the independence of its various components, F-logic can be viewed as a tool-kit for designing custom-tailored declarative languages. In fact, there is a more fine-grained division inside the logic. For instance, the semantics of scalar methods is independent from that of set-valued methods, and we can consider each type of methods as a separate tool. Likewise, we could require the oid's of classes and methods to be constants; method-name overloading can also be controlled in a number of ways (using sorts, for example). This tool-kit-like structure of F-logic is extremely helpful, as it lets us address different aspects of the logic in separation from each other, both theoretically and implementationally.

The last issue we would like to discuss is the relationship between predicate calculus and F-logic. In one direction, predicate calculus is a subset of F-logic and so the latter is more powerful than the former. However, it turns out that, in a sense, the two logics are equivalent.

Theorem 17.1 *There are mappings*

$$\begin{array}{lll} , & : \{\textit{F-formulas}\} & \longmapsto \{\textit{well-formed formulas of predicate calculus}\} \\ \Phi & : \{\textit{F-structures}\} & \longmapsto \{\textit{semantic structures of predicate calculus}\} \end{array}$$

such that

$$\mathbf{M} \models_F \psi \quad \textit{if and only if} \quad \Phi(\mathbf{M}) \models_{PC} , (\psi) \tag{28}$$

for any F-structure $\mathbf{M}$ *and any F-formula* ψ*; here "*$\models_F$*" and "*$\models_{PC}$*" denote logical entailment in F-logic and predicate calculus, respectively.* □

Proof: (Sketch) The proof is easy but tedious, so most of it is left as an exercise. The main idea is to introduce *new* predicates that encode the meaning of the methods for every arity and for every type of invocation.

Let $scalarNonInheritable_n$ be the predicate that is chosen to encode n-ary non-inheritable scalar data expressions. Similarly, we can choose $setNonInheritable_n$ to encode non-inheritable set-valued data expressions. Then we can split F-molecules into constituent atoms and represent each atom by an appropriate tuple in one of the above predicates. In addition, we shall need predicates $scalarType_n$ and $setType_n$ to encode scalar and set-valued signature expressions. For instance,

$$bob[jointWorks@phil \twoheadrightarrow X] \wedge empl[budget \Rightarrow integer]$$

can be represented as

$$setNonInheritable_1(jointWorks, bob, phil, X) \wedge scalarType_0(budget, empl, int)$$

Class membership and subclassing can be represented by appropriate binary predicates. To complete the encoding , , it remains to specify axioms that achieve the effect of the built-in features of F-structures. For instance, it will be necessary to write down a transitivity axiom for subclassing, type inheritance axioms, etc.

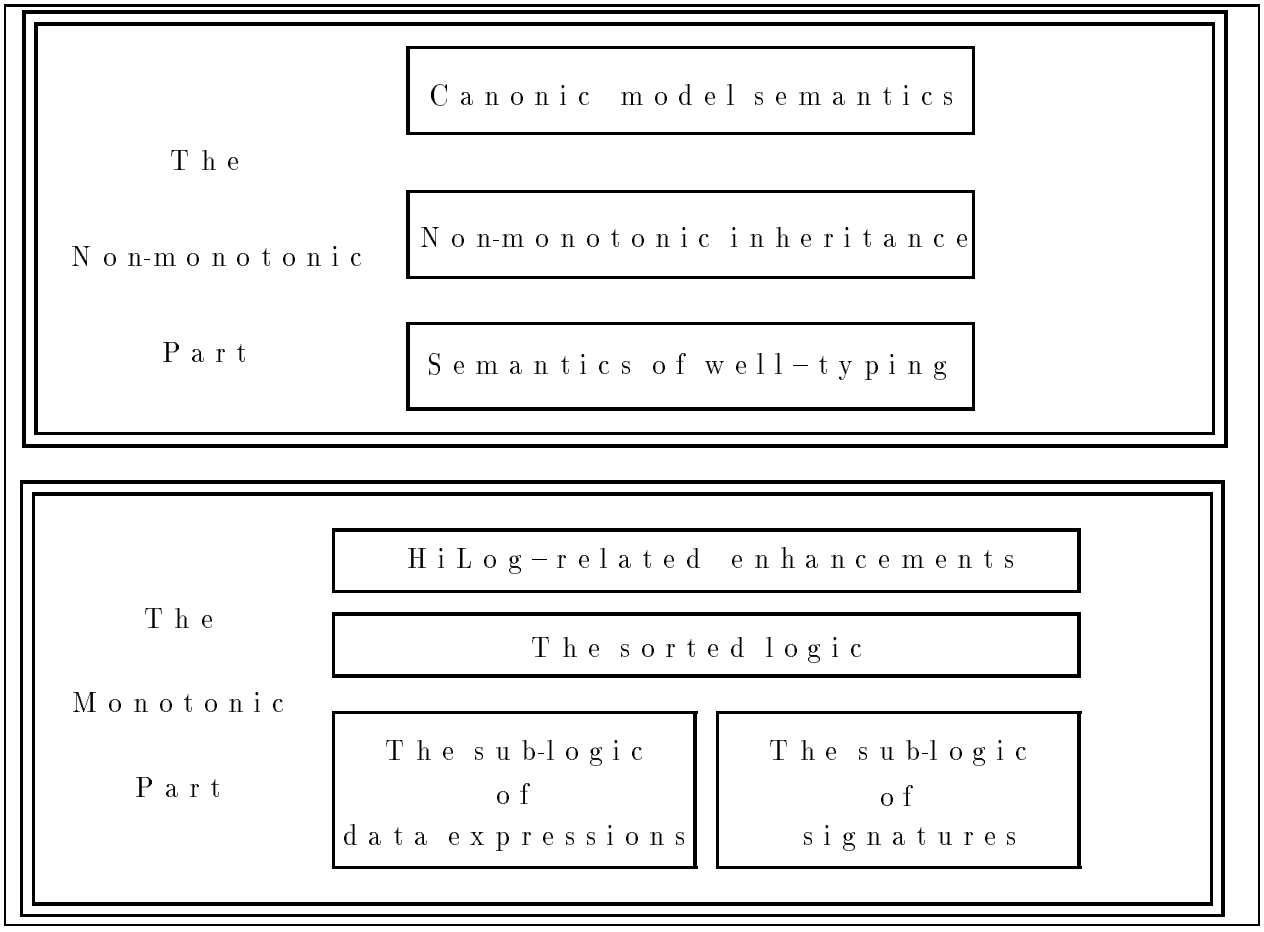

Figure 9: The Internal Structure of F-logic

semantics and the proof theory presented earlier. The glue between these two sublogics is provided by the well-typing semantic conditions of Section 12, in the non-monotonic part of the logic. Furthermore, notice that, as far as the monotonic logic is concerned, inheritable and non-inheritable data expressions are completely independent.

The third component is a technique for defining sorts in F-logic. This technique has been discussed in Section 16.1. The fourth component consists of ideas borrowed from HiLog [30], which extend the syntax of F-logic to allow variables and even complex terms to appear in the positions of predicate and function symbols, as discussed in Section 16.2.

The non-monotonic part of F-logic is also composed of a collection of distinct ideas. One of them is the canonic model semantics for F-programs with equality, as discussed in Section 11 and in Appendix A. The next two determine the semantics of non-monotonic inheritance and typing, both based on the idea of a canonic model. The notion of encapsulation also belongs here, since we view it simply as an elaborate policy of type-correctness. Note that, although Appendix A defines perfect models as an example of a canonic model semantics, our treatment of inheritance and typing is independent from this specific theory

language-engineering issues further, as they are beyond the scope of this paper.

16.2 HiLog-Inspired Extensions

HiLog [30] is a higher-order extension of predicate calculus that allows variables to range over the names of function and predicate symbols. Furthermore, these names themselves may have a rather complex structure and, in particular, the usual first-order terms (with variables) may appear in predicate positions. Many applications of HiLog are described in [30, 100, 84] and it is clear that extending F-logic in this direction is a good idea. In Section 11.4.1, we have seen one example (graph restructuring) where a combination of HiLog and F-logic may be useful.

We illustrate HiLog using the following example:

$$closure(Graph)(From, To) \leftarrow Graph(From, To)$$
$$closure(Graph)(From, To) \leftarrow Graph(From, Mid) \wedge closure(Graph)(Mid, To)$$

This HiLog program computes the closure of any binary relation that is passed to a predicate constructor, *closure*, as an argument. Here $Graph$ is a variable that ranges over predicates and $closure(Graph)$ is a *non-ground* term that denotes a parameterized family of predicates. For any given binary predicate, $graph$, the term $closure(graph)$ is another binary predicate that is true of a pair $\langle from, to \rangle$ precisely when $graph$ has a directed path connecting $from$ and to.

The ideas underlying HiLog are applicable to a wide range of logical languages, and the reader familiar with HiLog will have no difficulty to integrate them into F-logic by providing a semantics to HiLog-style terms. To this end, all that has to be done is to change the interpretation of id-terms from the one given in this paper to that in [30].

17 The Anatomy of F-logic

It is instructive to take a retrospective look back at the structure of F-logic and clarify the various relationships that exist between its different components. A careful examination would then reveal that most of the components of F-logic are independent from each other, both semantically and proof-theoretically. A high-level view of the internals of F-logic is depicted in Figure 9. In the figure, the logic is presented as a bag of ideas that can be classified into two main categories. The monotonic part of F-logic was presented in Sections 4 through Section 10. The non-monotonic part is comprised of a number of techniques, all based on the canonic model semantics. These techniques are described in Sections 11, 12, 14, and in Appendix A. The non-monotonic part of F-logic is based on the monotonic part, especially on its model theory. The proof theory presented in Section 10 is sound and complete for the monotonic part, but the non-monotonic extensions cannot have a complete proof theory. This is completely analogous to classical logic, where logic programs with stratified negation have no complete proof theory.

The two main components of the monotonic department are the logics of data expressions and the logic of signatures. These sublogics are independent from each other, as can be seen by inspecting the

16.1 Sorted F-logic and its Uses

In some cases it may be necessary to distinguish between set-valued and scalar attributes at the syntactic level. (Previously, we could distinguish them only at the meta-level, via the notion of type correctness). In other cases, we may need to designate certain objects as "true" individual objects, namely, objects that can never play the role of a class. For instance, objects that represent people, such as *john*, *mary*, etc., may be in this category. Using sorts to separate things that should not be mingled in the same domain is an old idea. The advantage of having sorts in addition to types is that well-formedness with respect to sorts is easier to verify, usually much easier.

To illustrate how this idea can be applied to F-logic, suppose we wanted to keep a distinction between classes and individuals and yet be able to manipulate both kinds of objects in a uniform way. To this end, we could separate all variables and function symbols into three categories with different name spaces. For instance, to represent individuals, we could use id-terms whose outermost symbol's name begins with a "!"; to represent classes, we may use id-terms beginning with a "♯"-symbol; and id-terms that begin with any other legal symbol (*e.g.*, a letter, as it was so far) can be used to represent *any* object, an individual or a class. These latter symbols allow accessing both kinds of objects uniformly, without duplicating program clauses to deal with each sort separately.

Semantically, sorts are accommodated in a standard way. The domain, U, of an F-structure would now consist of two disjoint parts: $U^{\sharp}$—to interpret class-objects; and $U^{!}$—for individual objects. Variables beginning with a ♯-symbol will then range over the subdomain of classes, while !-variables will be restricted to the domain of individuals. All other variables will be allowed to range over the entire domain, U. Since we wish to allow class-objects to be constructed out of both, class-objects and individual-objects, a function symbol beginning with a "♯" will have the type $U^n \mapsto U^{\sharp}$, for a suitable n, while a function symbol beginning with a "!"-sign will have the type $U^n \mapsto U^{!}$.

Alternatively, the same effect can be achieved by introducing a pair of new unary predicates, $class(X)$ and $individual(X)$, where *class* and *individual* are true precisely of those ground id-terms whose outermost symbol starts with a "♯" and "!", respectively. In addition, rules and queries should be relativized as follows: For every individual id-term, $!T$, in the body of a rule, $head \longleftarrow body$ (or in a query $?- query$), add $individual(!T)$ as a conjunct, obtaining $head \longleftarrow body \wedge individual(!T)$ (resp., $?- query \wedge individual(!T)$). Likewise, for every id-term that represents a class, $\sharp S$, add $class(\sharp S)$ as a conjunct in the body of the rule or the query.

Next we could further restrict logical formulas, such as signatures and is-a assertions. For instance, in $T[Mthd@V_1, ..., V_n \Rightarrow W]$, we may require T, V_1, ..., V_n, W to represent classes, (*i.e.*, all signatures will actually look like $\sharp T[Mthd@\sharp V_1, ..., \sharp V_n \Rightarrow \sharp W]$). Likewise, for is-a assertions of the form $O : Cl$ and $Cl :: CL'$ we may require that Cl and Cl' will be classes.

The above ideas effectively lead to a 3-sorted logic on top of the basic F-logic. However, the utility of sorts is not limited to distinguishing individuals or methods from other objects. In principle, we may let users define their own naming conventions. In each case, this will add a new sort of symbols to the logic, to harbor the oid's of objects the user wishes to distinguish. We shall not discuss these

equation

$$current(chip123) \doteq version(mary, version(john, version(john, chip123)))$$

and delete any other equation of this form that might have been previously asserted.

Navigation through the versions is quite easy. For instance, to access to the previous version of the chip, one can use the following query:

$$?- \ (current(chip123) \doteq version(Whoever, PrevVersion)) \wedge PrevVersion[\ldots] \wedge \ldots$$

Another way to organize versions in F-logic is to define a method, *version*, that returns appropriate versions, depending on the argument. To access a version of $chip123$, one could use

$$?- \ chip123[version@arg \rightarrow Version] \wedge Version[\ldots]$$

where arg is a suitable encoding of the desired version. Yet another way to do versioning is to parameterize attributes with version id's, as suggested in [45].

15.5 Path Expressions

Many calculus-like object-oriented languages (*e.g.*, [102, 14]) use syntactic constructs known as *path expressions*. For instance, to select employees working in "CS" departments, one could use a path expression $X.worksFor.dname \doteq$"CS" instead of $X[worksFor \rightarrow D[dname \rightarrow$"$CS$"$]]$.

The idea of path expressions was significantly extended in [47] and has become a basis of an object-oriented extension of SQL. In the syntax of [47], the expression (ix) of Figure 4 can be written thus: $X.worksFor.dname$["CS"]. Here, "CS" plays the role of a *selector* that selects sequences of objects x, y, z such that $x.worksFor = y$, $y.dname = z$, and $z =$ "CS".

Although path expressions are not explicitly part of F-logic syntax, they can be added as a syntactic sugar to help reduce the number variables in F-programs (and also because this notation is more familiar to database users). For instance, we could write:

$$?- \ X : empl \wedge X.name[N] \wedge X.friends.name[\text{"}Bill\text{"}] \wedge X.worksFor.dname[\text{"}CS\text{"}]$$

instead of the bulkier query:

$$?- \ X : empl[name \rightarrow N; friends \twoheadrightarrow F[name \rightarrow \text{"}Bill\text{"}]; worksFor \rightarrow D[dname \rightarrow \text{"}CS\text{"}]]$$

Formal translation of path expressions (as defined in [47]) into F-logic is left to the reader as an easy exercise.

16 Extensions to F-logic

In this section, we explore two enhancements of the logic: the introduction of sorts and extending F-logic in the direction of HiLog [30].

where, as in Section 6, *p-tuple* is a function symbol specificly chosen to represent "invented objects" of class p. In other words, invention atoms of [44] are essentially *value-based* P-molecules of F-logic.[26]

The "$*$"-style don't-care symbols can be useful for modeling value-based objects, such as relations with mnemonically meaningful attribute names. For instance, we could represent the SUPPLIER relation via a class, *suppl*, declared as follows:

$$suppl[name \Rightarrow string;\ parts \Rightarrow\!\!\!> part;\ addr \Rightarrow string]$$

Then we could define the contents of that relation as follows:

$$*:suppl[name \rightarrow \text{"XYZ Assoc."};\ parts \twoheadrightarrow all\text{-}in\text{-}one;\ addr \rightarrow \text{"Main St., USA"}]$$
$$*:suppl[name \rightarrow \text{"Info Ltd."};\ parts \twoheadrightarrow know\text{-}how;\ addr \rightarrow \text{"P.O. Box OO"}]$$
$$\ddots$$

Here, the symbol "$*$" is used as a short-hand notation intended to relieve programmers from the tedium of specifying oid's explicitly (which are $suppl\text{-}tuple(\text{"}XYZ\ Assoc.\text{"}, \text{"}Main\ St.\ USA\text{"})$ and $suppl\text{-}tuple(\text{"}Info\ Ltd.\text{"}, \text{"}P.O.\ Box\ OO\text{"})$ in the above).

It should be clear that the semantics of "$*$"-style objects is different from the semantics of "$\_$"-style objects. For instance, "$*$"-style objects are entirely *value-based*, as their id's are uniquely determined by their structure. Therefore, an "$*$"-style object can live its own, independent life. Its oid is replaced by "$*$" simply because this oid is uniquely determined by the values of the scalar attributes of the object, making the explicit mention of the id redundant. In comparison, "$\_$"-objects are *not* value-based. The identity of such an object depends mostly on the syntactic position occupied by this object inside another molecule. What the two don't-care styles share, though, is that in both cases the explicit mention of the object identity is redundant, as it is uniquely determined from the context or by the object structure.

15.4 Version Control

Although version control is not an intrinsically object-oriented feature, it has been often associated with object-oriented databases because of the importance of versioning in several "classical" object-oriented applications, such as computer aided design. Conceptually, version control has two distinct functionalities: creation of new versions and navigation among the existing versions. As an example, we outline one of the several possible schemes for realizing version control in F-logic.

Suppose Mary and John are working cooperatively on a computer chip, denoted $chip123$. Each time Mary makes a modification to a version, v, of the chip, she creates a new object, $version(mary, v)$. Similarly, John constructs versions of the form $version(john, v)$. Thus, all versions of the chip have the form $version(p_n, \ldots version(p_1, chip123)\ldots)$, where each p_i is either $mary$ or $john$.

Both Mary and John access the current version of $chip123$ via the name $current(chip123)$. To turn a version, say $version(mary, version(john, version(john, chip123)))$, into a current one, they assert an

[26] However, ILOG is not an entirely value-based language, since there is a way (albeit indirect) to create multiple copies of distinct objects with the same contents.

First, note that (i) above is already legal in data expressions. Also, (ii) is "almost" legal: it is a generalization of the set-construct, where previously we allowed only sets of id-terms on the right-hand side in data expressions.

The next question is the semantics of values introduced in this way. The answer depends on the intended usage of values. If they are taken too literally, as terms assignable to logical variables, this could jeopardize the proof theory of our logic. Indeed, by (i), variables are values and so they can unify with other values. But then the term $obj[attr \rightarrow X]$ should be unifiable with, say,

$$obj[attr \twoheadrightarrow \{a, b, c\}\,]$$

where X unifies with the set $\{a, b, c\}$. In other words, (i) entails that variables can range over sets, thereby rendering the semantics second-order in the sense of [30].

Fortunately, the experience with object-oriented database systems, such as Orion [53] and O_2 [63], suggests that complex values are used in fairly restricted ways—primarily as *weak entities* of the Entity-Relationship Approach. In other words, they are used as "second-rate objects," accessible only through other objects.

This leads to the conclusion that values are used not for their own sake, but rather for convenience, to let the user omit oid's when they are immaterial. To an extent, this is similar to Prolog's don't-care variables represented by underscores. By analogy, we could permit the use of don't-care object id's. For instance, we can re-cast the previous example as follows:

$$eiffelTower[name \rightarrow \text{``}Eiffel\ Tower\text{''};\ address \rightarrow \_[city \rightarrow paris;\ street \rightarrow champDeMars]\,]$$

It remains to decide on a suitable semantics of the don't-care symbol, "_". Let

$$T[\,ScalM \circledcirc S_1, ..., S_n \rightarrow \_[\ldots]\,]$$

be a term with a don't-care symbol. It's meaning can be given through the following encoding:

$$T[\,ScalM \circledcirc S_1, ..., S_n \rightarrow val_n(T, ScalM, S_1, ..., S_n)[\ldots]\,]$$

where val_n is an $(n+2)$-ary new function symbol, specifically chosen for this kind of encoding. It is not accidental that we have given no interpretation to don't-care symbols inside the set-construct, as in

$$T[\,SetM \circledcirc R_1, ..., R_m \twoheadrightarrow \{..., \_[\ldots], ...\}\,]$$

The meaning and the utility of "_" in this context is unclear to us. However, below, we introduce a don't-care symbol, "$*$", with a different semantics, and this kind of oid may well occur inside sets.

The need for don't-care oid's was also emphasized in ILOG [44], where a special "$*$"-notation was introduced for this purpose. An *invention atom*, $p(*, t_1, ..., t_n)$, of ILOG is semantically close to an F-molecule of the form

$$p\text{-}tuple(t_1, \ldots, t_n) : p[arg_1 \rightarrow t_1;\ \ldots;\ arg_n \rightarrow t_n]$$

However, when the user poses a query, such as

$$?- \; john[children \twoheadrightarrow X] \tag{27}$$

and X gets no bindings, it is unclear whether this is because John has no children or because this attribute has a null value. To verify this, the user will have to ask one more query: $?- \; john[children \twoheadrightarrow \{\}]$.

Clearly, asking two queries to find out one simple thing is a questionable practice, and it is highly desirable that a single query, (27), will suffice. One easy way to fix the problem is to introduce a special constant, $\emptyset$, into every F-logic language, along with the following axiom schemata:

$$X[\, M @ A_1, ..., A_n \twoheadrightarrow \emptyset \,]$$
$$X[\, M @ A_1, ..., A_n \bullet\!\!\twoheadrightarrow \emptyset \,]$$

where $n = 0, 1, 2, ...$ Technically speaking, these axioms ensure that methods never return empty sets. However, *conceptually*, empty sets can be represented via sets that contain $\emptyset$ alone.

15.3 Complex Values

Several recent works [15, 3, 63] have suggested that pure object-based languages may be burdensome when identity of certain objects is immaterial to the programmer. We, too, have made this point in Section 6, advocating the use of predicates on a par with objects. However, it may sometimes be convenient to go beyond simple predicates and allow object-like structures that have no associated object id. Such structures are called *complex values* [15, 3, 63]. For instance, adapting an example from [63] we could write:

$$eiffelTower[name \rightarrow \text{“}Eiffel\ Tower\text{”}; \; address \rightarrow [city \rightarrow paris; \; street \rightarrow champDeMars] \,]$$

Here $[city \rightarrow paris; \; street \rightarrow champDeMars]$ is a “complex value” representing an address. In O_2 [63], such constructs have no identity and are distinct from objects. The rationale is that the user does not have to shoulder the responsibility for assigning id's to objects that are not going to be accessed through these id's.

In the spirit of [3, 63], we can try to adapt the concept of complex values to F-logic as follows:

(i) Any id-term is a *value*;

(ii) $\{Val_1, \ldots, Val_n\}$, is a value, provided that each Val_i is a value;

(iii) $[\ldots, \; Attr \rightsquigarrow Val, \ldots]$ is a value, where
$Attr$ is an id-term, Val is a value, and $\rightsquigarrow$ is one of the four arrows
that may occur in data expressions.

To account for complex values, the definition of data expressions in Section 4 should be changed to allow complex values (rather than just oid's) to appear on the right-hand side of data expressions.

Now, let sup_1, ..., sup_n denote all the classes where *obj* belongs and to which the method *mthd* applies. Then *obj* will inherit the methods $mthd(sup_1)$, ..., $mthd(sup_n)$. This time, however, there is no conflict between these methods, due to parameterization. So, *obj* can invoke the method *mthd* as defined in any of its superclasses, *e.g.*,

$$?- \quad obj[mthd(sup_7)@a, b \rightarrow Z]$$

This is analogous to the use of scope resolution operators in object-oriented languages.

15 Other Issues in Data Modeling

While F-logic has a wide range of object-oriented concepts represented directly in its semantics, several other ideas have been left out. These include complex values, path expressions, and others. In this section, we propose ways to model these missing concepts using the rich arsenal of the built-in features available in F-logic.

15.1 Existing and Non-existent Objects

It is easy to verify that $\mathbf{P} \models t[]$ holds for any id-term, t, and any program, $\mathbf{P}$. In this sense, in F-logic, any object exists as long as it can be named. On the other hand, in databases and other object-oriented systems, it is a common practice to distinguish between existing objects and those that do not exist. To come into existence, even an empty object must first be created, which seems to be in stark contrast to F-logic where empty objects, such as $t[]$ above, exist all the time.

Fortunately, the concept of "existence" can be easily expressed as a property of an object, using a special attribute, say, *exists*. For instance, if $t[exists \rightarrow true]$ is in the canonical model of $\mathbf{P}$ then the object is said to exist. Otherwise, it does not.

Then, a special preprocessor can modify every clause in $\mathbf{P}$ in such a way that every object molecule that contains a data expression will be augmented with another data expression, $exists \rightarrow true$. Queries should be modified accordingly. For instance, the query $?- X$ will be changed into $?- X[exists \rightarrow true]$, and so only the id's of the existing objects will be retrieved.

15.2 Empty Sets vs. Undefined Values

The semantics of set-valued methods distinguishes between the cases when the value of a method is an empty set and when the method is undefined. Although in some cases the user may not care about this distinction, this difference is important in many other cases. For instance, asserting $john[children \twoheadrightarrow \{\}]$ means that *john* has no children (provided that there are no other statements of the form $john[children \rightarrow \rightarrow xyz]$). In contrast, when $john[children \twoheadrightarrow \{\}]$ does not hold, this means that it is *unknown* whether *john* has children, as method undefinedness is essentially a form of null values familiar from classical database theory.

override the default. However, notice that overriding has taken place only for some *specific* arguments, while the default definition is still being used for other arguments (*e.g.*, when $Crs = vlsi$).

This property of F-logic inheritance is called *pointwise* inheritance. As we have seen, pointwise overriding provides a great deal of flexibility by allowing to modify methods over specific subsets of arguments. This should be contrasted with most other languages where methods have to be overwritten in their entirety or not overwritten at all.

Total Method Overriding

In addition to pointwise overriding, it is easy to override methods in their entirety, as in most object-oriented languages. Consider the following example:

$$\begin{array}{l} bob : instructor \\ instructor[gradingMethod \bullet\!\!\rightarrow defaultGrading] \\ X[defaultGrading @ Stud, Crs \rightarrow G] \longleftarrow X : instructor \wedge \text{definition-4} \\ bob[gradingMethod \rightarrow bobGrading] \\ bob[bobGrading \rightarrow G] \longleftarrow \text{definition-5} \\ X[grade @ Stud, Crs \rightarrow G] \longleftarrow X : instructor[gradingMethod \rightarrow M;\ M @ Stud, Crs \rightarrow G]. \end{array} \tag{26}$$

In this example, the value of $gradingMethod$ is the name of a method to be used for grade computation. This method is defined (in the third clause) for each *instructor*-object. The fourth clause specifies the name of the grading method that *bob* prefers to use. This method is defined on the object *bob* using the fifth clause.

The last clause, then, defines *grade* using the method M obtained from the $gradingMethod$ attribute. Normally, this value would be inherited from the class *instructor* and will be $defaultGrading$. However, the value of the $gradingMethod$ attribute is overwritten on object *bob*. Therefore, when X is bound to *bob*, M will be bound to $bobGrading$ and so *grade* will behave exactly like $bobGrading$. In contrast, if *mary* is also an instructor, but she did not override the attribute $gradingMethod$, the value for M in the last clause will be inherited from *instructor* and will thus be $defaultGrading$. Therefore, when X is bound to *mary*, the method *grade* will behave exactly like $defaultGrading$.

User-Controlled Inheritance

The semantics of inheritance described in this section is non-deterministic, *i.e.*, in case of an inheritance conflict, the system will fire one of the active triggers non-deterministically. In some cases, however, the user may want to have more control over trigger-firing. This approach is common in many programming languages, such as Eiffel [70] or C++ [89], where the programmer has to resolve inheritance conflicts explicitly.

User-defined inheritance can be expressed in F-logic as follows. To resolve a conflict, say, in a binary method, *mthd*, we can first parameterize *mthd* by defining a family of methods, $mthd(Class)$:

$$Class[mthd(Class) @ X, Y \bullet\!\!\rightarrow Z] \longleftarrow Class[mthd @ X, Y \bullet\!\!\rightarrow Z].$$

object	**L**	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$
p	$attr_1 \bullet\!\!\twoheadrightarrow c$	$attr_1 \bullet\!\!\twoheadrightarrow c$	$attr_1 \bullet\!\!\twoheadrightarrow c$	$attr_1 \bullet\!\!\twoheadrightarrow c$
t	$attr_1 \bullet\!\!\twoheadrightarrow d$	$attr_1 \bullet\!\!\twoheadrightarrow d$	$attr_1 \bullet\!\!\twoheadrightarrow d$	$attr_1 \bullet\!\!\twoheadrightarrow d$
a	—	$attr_1 \twoheadrightarrow c$	$attr_1 \twoheadrightarrow d$	$attr_1 \twoheadrightarrow c$
a	—	$attr_2 \twoheadrightarrow e$		$attr_2 \twoheadrightarrow e$
a	—	—	—	$attr_1 \twoheadrightarrow d$
is-a	$a : p$	$a : t,\ t :: p$	$a : t,\ t :: p$	$a : t,\ t :: p$

Among the above models, only $\mathbf{M}_1$ is ι-canonical; it is obtained from **L** by firing the previously defined triggers τ and κ.

The subtle point here is that if $a : t$ and $t :: p$ were true in the initial model, **L**, the existence of $t[attr_1 \bullet\!\!\twoheadrightarrow d]$ on the path $a : t :: p$ would have blocked the inheritance of $attr_1 \bullet\!\!\twoheadrightarrow c$ from p to a. However, blocking an inheritance path *after* the inheritance step is performed does not undo inheritance acts performed earlier.

14.3 Modeling Other Overriding Strategies

In this section, we illustrate various ways in which F-logic inheritance can model overriding strategies used in other languages.

Pointwise Overriding

Suppose that the class *instructor* provides a default definition for the method *grade* used to compute students' grades in different courses:

$$\begin{aligned} instructor[grade@Stud, vlsi \bullet\!\!\rightarrow G] &\longleftarrow \text{definition-1} \\ instructor[grade@Stud, db \bullet\!\!\rightarrow G] &\longleftarrow \text{definition-2} \\ &\ddots \end{aligned}$$

Any instructor, say *bob* (assuming $bob : instructor$ holds), can access this method by asking queries:

$$?- bob[grade@mary, vlsi \rightarrow G]$$

Suppose $instructor[grade@mary, vlsi \bullet\!\!\rightarrow 90]$ holds in class *instructor*. Then $grade@mary, vlsi \bullet\!\!\rightarrow 90$ will be inherited by *bob*, deriving the atom $bob[grade@mary, vlsi \rightarrow 90]$. Similar inheritance will take place for *db* and other courses.

Suppose now that instructors are allowed to modify the default grade computation algorithm by defining their own. Thus, for instance, *bob* may define his own policy in the database course:

$$bob[grade@Stud, db \rightarrow G] \longleftarrow \text{definition-3}$$

This would provide an explicit definition of *grade* in the object *bob* when this method is invoked with arguments *stud* and *db*, where *stud* represents an arbitrary student. Thus, this explicit definition will

object	$\mathbf{L}$	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$
q	$attr \bullet\!\!\twoheadrightarrow a$	$attr \bullet\!\!\twoheadrightarrow a$	$attr \bullet\!\!\twoheadrightarrow a$	$attr \bullet\!\!\twoheadrightarrow a$
p	—	$attr \twoheadrightarrow a$	—	$attr \twoheadrightarrow a$
r	—	$attr \twoheadrightarrow b$	$attr \twoheadrightarrow a$	$attr \twoheadrightarrow a$
r	—	—	—	$attr \twoheadrightarrow b$
is-a	$p:q,\ r:q$	$p:q,\ r:q$	$p:q,\ r:q$	$p:q,\ r:q$

Both $\mathbf{M}_1$ and $\mathbf{M}_3$ are ι-canonical models. Observe how the rule application that produced $\mathbf{M}_1$ from $I^{\tau}_{\mathbf{P}}(\mathbf{L})$ had deactivated the trigger, κ. On the other hand, firing κ did not prevent the rule from firing.

The existence of two ι-canonic models for the above program may appear discomforting, because a case can be made that trigger τ should have been fired before trigger κ. Indeed, firing the former obviates the need to fire the latter and, in that sense, $\mathbf{M}_1$ is a "smaller" model than $\mathbf{M}_3$. This can be seen even more clearly if we replace "$\twoheadrightarrow$" and "$\bullet\!\!\twoheadrightarrow$" in (25) with "$\rightarrow$" and "$\bullet\!\!\rightarrow$", respectively. In this case, the model $\mathbf{M}_3$ (with double-headed arrows turned into the single-headed ones) would contain the atom $a \doteq b$ and thus also $r[attr \rightarrow b]$. This turns $\mathbf{M}_1$ into a strict *subset* of $\mathbf{M}_3$, further fueling our suspicion about the wisdom of keeping $\mathbf{M}_3$ as a canonic model.

The problem with premature firing of triggers (as in the case of κ above) can be solved by ordering active triggers in certain ways. This issue will be discussed separately, in [49].

The following example illustrates the phenomenon of inheritance in dynamically changing IS-A hierarchies. Let the program, $\mathbf{P}$, be:

$$
\begin{array}{ll}
a:p & p[attr_1 \bullet\!\!\twoheadrightarrow c] \\
a:t \longleftarrow a[attr_1 \twoheadrightarrow c] & t[attr_1 \bullet\!\!\twoheadrightarrow d; attr_2 \bullet\!\!\twoheadrightarrow e] \\
t::p \longleftarrow a[attr_1 \twoheadrightarrow c] &
\end{array}
$$

Let $\mathbf{L}$ be the minimum model of $\mathbf{P}$. Then $\tau = \langle attr_1, \bullet\!\!\twoheadrightarrow, a:p \rangle$ is the only active trigger in $\mathbf{L}$. Firing τ causes $a:t$ and $t::p$ to be derived. This, in turn, activates the trigger $\kappa = \langle attr_2, \bullet\!\!\twoheadrightarrow, a:t \rangle$, and $a[attr_2 \twoheadrightarrow e]$ is derived by inheritance. Note that there is no inheritance of $attr_1 \bullet\!\!\twoheadrightarrow d$ from t to a, since $attr_1$ is already defined on a.

The most interesting aspect of this example is that the very act of inheritance alters the IS-A hierarchy on which it depends. Moreover, the derivation of $a:t$ and $t::p$ propels the fact $t[attr_1 \bullet\!\!\twoheadrightarrow d]$ with an inheritable property right in the middle of the path through which a had inherited $attr_1 \bullet\!\!\twoheadrightarrow c$ from p. This issue was already discussed in connection with Example (24) earlier, where we had argued that the above inheritance step should not be undone, despite the changes in the IS-A hierarchy.

To see how this dynamic situation is taken care of by our semantics, consider the following table that presents the relevant parts of several models of interest:

We thus see that both, $\mathbf{M}_2$ and $\mathbf{M}_3$, are obtained from $\mathbf{M}_1$, albeit by firing different triggers. These triggers are associated with the same attribute, *policy*, and the same recipient object, *nixon*, but with different contributing superclasses, *quaker* and *republican*. For this reason, inheritance steps obtained by firing each trigger separately cannot be combined, because firing of one trigger deactivates the other, and vice versa. The existence of a pair of triggers with these properties is a formal explanation for the phenomenon of multiple inheritance.

The following result is a direct consequence of the definitions:

Lemma 14.4 *Let* $\mathbf{P}$ *be a Horn F-program and let* $\mathbf{I}$ *be an H-structure for* $\mathbf{P}$*. Let, further,* τ *be an active trigger in* $\mathbf{I}$*. Then* $TI^{\tau}_{\mathbf{P}}(\mathbf{I}) = (T_{\mathbf{P}} \uparrow \omega)(I^{\tau}_{\mathbf{P}}(\mathbf{I}))$*, where* $T_{\mathbf{P}} \uparrow \omega$ *denotes the countably-infinite iteration of* $T_{\mathbf{P}}$*.* □

This lemma leads to a result, below, that sheds light on the computational properties of ι-canonical models of Horn programs. First, we need to define iterations of the operator $TI_{\mathbf{P}}$. Since $TI^{\tau}_{\mathbf{P}}$ is monotone with respect to H-structures that have τ as an active trigger, iterations can be defined in a standard way, where we only have to take care of the non-deterministic selection of triggers.

- If $\alpha = 0$, then $(TI^{\tau}_{\mathbf{P}} \uparrow \alpha)(\mathbf{I}) = \mathbf{I}$, for any trigger τ.
- If α is a non-limit ordinal, $(TI_{\mathbf{P}} \uparrow \alpha)(\mathbf{I}) = TI^{\tau}_{\mathbf{P}}[(TI_{\mathbf{P}} \uparrow (\alpha - 1))(\mathbf{I})]$, for any trigger τ that is active in $(TI_{\mathbf{P}} \uparrow (\alpha - 1))(\mathbf{I})$. That is, the result of the transition from $(TI_{\mathbf{P}} \uparrow (\alpha - 1))(\mathbf{I})$ to $(TI_{\mathbf{P}} \uparrow \alpha)(\mathbf{I})$ is non-deterministic, as it depends on the trigger selected.
- If α is a limit ordinal, $(TI_{\mathbf{P}} \uparrow \alpha)(\mathbf{I}) = \bigcup_{\beta<\alpha}(TI_{\mathbf{P}} \uparrow \beta)(\mathbf{I})$.

Proposition 14.5 *Let* $\mathbf{P}$ *be a Horn F-program and* $\mathbf{L}$ *be its minimum model. Then* $\mathbf{P}$ *has at least one* ι*-canonical model, which can be obtained by iterating* $TI_{\mathbf{P}}$ *using an appropriate sequence of triggers and taking* $\mathbf{L}$ *as the initial H-structure. Conversely,* $(TI_{\mathbf{P}} \uparrow \omega)(\mathbf{L})$ *is an* ι*-canonical model of* $\mathbf{P}$*, where* ω *is the first infinite ordinal.* □

The next F-program, $\mathbf{P}$, illustrates one of the many ways in which inheritance can interact with deduction:

$$\begin{array}{ll} p : q & q[attr \bullet\!\!\rightarrow\!\!\!\!\rightarrow a] \\ r : q & r[attr \rightarrow\!\!\!\!\rightarrow b] \longleftarrow p[attr \rightarrow\!\!\!\!\rightarrow a]. \end{array} \tag{25}$$

There are two triggers, $\tau = \langle attr, \bullet\!\!\rightarrow\!\!\!\!\rightarrow, p : q\rangle$, and $\kappa = \langle attr, \bullet\!\!\rightarrow\!\!\!\!\rightarrow, r : q\rangle$. Observe that applying τ will activate the last clause. Let $\mathbf{L}$ be the minimum model of $\mathbf{P}$, and let $\mathbf{M}_1, \mathbf{M}_2$ be the two models that emerge after firing one of the triggers in $\mathbf{L}$, *i.e.*, $TI^{\tau}_{\mathbf{P}}(\mathbf{L}) = \mathbf{M}_1$ and $TI^{\kappa}_{\mathbf{P}}(\mathbf{L}) = \mathbf{M}_2$. It is easy to verify, that applying the first trigger, τ, deactivates κ, but firing κ does not deactivate τ. Therefore, there is no trigger to fire in $\mathbf{M}_1$ and no further inheritance is possible. However, τ can still be fired in $\mathbf{M}_2$, leading to another model, $\mathbf{M}_3 = TI^{\tau}_{\mathbf{P}}(\mathbf{M}_2)$. The following table shows the relevant parts of these models:

Note that $TI^{\tau}_{\mathbf{P}}$ is monotonic with respect to H-structures that have τ as a trigger, since $I^{\tau}_{\mathbf{P}}$ is also monotonic on such structures.

We are now ready to define the notion of canonical models that take inheritance into account.

Definition 14.3 (Canonical Models with Inheritance) Let $\mathbf{P}$ be a Horn F-program. A model $\mathbf{M}$ of $\mathbf{P}$ is *inheritance-canonical* (or *ι-canonical*) if and only if $\mathbf{M}$ is a minimal element with respect to $\ll^{inh}$ in the set $\{\mathbf{N} \mid \mathbf{N} \ll^{inh} \mathbf{L}\}$ of H-models, where $\mathbf{L}$ is the unique minimum model of $\mathbf{P}$ (which exists, since $\mathbf{P}$ is Horn, by assumption). □

We shall now illustrate the notions of one-step inheritance and of ι-canonical model on a number of examples. The following two examples are derived from the Nixon's Diamond example.

Consider the following program, $\mathbf{P}$:

$$nixon : republican \qquad\qquad republican[policy \bullet\!\!\rightarrow hawk]$$

This states that *nixon* belongs to the class *republican* and that, generally, republicans are hawks. $\mathbf{P}$ has the following models of interest that are summarized below:

	$\mathbf{M}_1$	$\mathbf{M}_2$
object		
republican	$policy \bullet\!\!\rightarrow hawk$	$policy \bullet\!\!\rightarrow hawk$
nixon	—	$policy \rightarrow hawk$
is-a	$nixon : republican$	$nixon : republican$

Here $\mathbf{M}_1$ is a minimal model of $\mathbf{P}$, but it is not ι-canonical. To see this, note that $\mathbf{M}_2 = I^{\tau}_{\mathbf{P}}(\mathbf{M}_1)$, where $\tau = \langle policy, \bullet\!\!\rightarrow, nixon : republican\rangle$, and that, in fact, $\mathbf{M}_2$ is derived from $\mathbf{M}_1$ by one inheritance step. It is easy to verify that, furthermore, $\mathbf{M}_2$ is an ι-canonical H-model of the program. If, however, $\mathbf{P}$ contained a clause $nixon[policy \rightarrow pacifist]$, then no inheritance could have taken place, because the more specific value, *pacifist*, overrides the inheritance of the value *hawk* from the class *republican*.

The next example illustrates multiple inheritance. Let the program $\mathbf{P}$ be:

$$nixon : republican \qquad\qquad republican[policy \bullet\!\!\rightarrow hawk]$$
$$nixon : quaker \qquad\qquad quaker[policy \bullet\!\!\rightarrow pacifist]$$

The following H-models are of interest here:

	$\mathbf{M}_1$	$\mathbf{M}_2$	$\mathbf{M}_3$
object			
republican	$policy \bullet\!\!\rightarrow hawk$	$policy \bullet\!\!\rightarrow hawk$	$policy \bullet\!\!\rightarrow hawk$
quaker	$policy \bullet\!\!\rightarrow pacifist$	$policy \bullet\!\!\rightarrow pacifist$	$policy \bullet\!\!\rightarrow pacifist$
nixon	—	$policy \rightarrow hawk$	$policy \rightarrow pacifist$
is-a	$nixon : republican$	$nixon : republican$	$nixon : republican$
	$nixon : quaker$	$nixon : quaker$	$nixon : quaker$

These models are related as follows: $I^{\tau}_{\mathbf{P}}(\mathbf{M}_1) = \mathbf{M}_2$ and $I^{\kappa}_{\mathbf{P}}(\mathbf{M}_1) = \mathbf{M}_3$, where $\tau = \langle policy, \bullet\!\!\rightarrow, nixon : republican\rangle$ and $\kappa = \langle policy, \rightarrow, nixon : quaker\rangle$. Furthermore, it is easy to check that $\mathbf{M}_2$ and $\mathbf{M}_3$ are both ι-canonical models of $\mathbf{P}$, while the model $\mathbf{M}_1$ is minimal but not ι-canonical.

The intention of firing a trigger of the form $\tau = \langle m, \bullet\!\!\twoheadrightarrow, obj : cl, a_1, ..., a_n \rangle$, is to inherit to the recipient object, obj, the result of the application of m on the source-class. To capture this idea, we introduce an operator of *firing* an active trigger, τ, on an H-structure, $\mathbf{I}$. This operator is denoted $I^{\tau}_{\mathbf{P}}$ and is defined as follows:[25]

$$I^{\tau}_{\mathbf{P}}(\mathbf{I}) \quad = \quad \mathbf{I} \ \cup \ \{obj[m@a_1, ..., a_n \twoheadrightarrow v] \mid \ v \in \mathbf{I}(cl[m@a_1, ..., a_n], \bullet\!\!\twoheadrightarrow)\},$$

Note that since obj is a member of cl, the inheritable properties of cl are passed down to obj as non-inheritable properties. On the other hand, if obj were a subclass of cl, *i.e.*, if the trigger had the form $\kappa = \langle m, \bullet\!\!\twoheadrightarrow, obj :: cl, a_1, ..., a_n \rangle$, then the properties of cl would have been passed down to obj as inheritable properties:

$$I^{\kappa}_{\mathbf{P}}(\mathbf{I}) \quad = \quad \mathbf{I} \ \cup \ \{obj[m@a_1, ..., a_n \bullet\!\!\twoheadrightarrow v] \mid \ v \in \mathbf{I}(cl[m@a_1, ..., a_n], \bullet\!\!\twoheadrightarrow)\},$$

For scalar invocations, this operator is defined similarly, by replacing $\twoheadrightarrow$ with $\rightarrow$ and $\bullet\!\!\twoheadrightarrow$ with $\bullet\!\!\rightarrow$.

Note that $I^{\tau}_{\mathbf{P}}$ is a monotonic operator on the H-structures that have τ as an active trigger. Once a trigger is fired, program rules that were previously inactive may become applicable and more facts can be deduced. For instance, consider the following program $\mathbf{P}$:

$$\begin{array}{l} a : b \\ b[attr \bullet\!\!\twoheadrightarrow c] \\ a[attr \twoheadrightarrow d] \Longleftarrow a[attr \twoheadrightarrow c] \end{array}$$

The minimal model here consists of $a : b$ and $b[attr \twoheadrightarrow c]$. Firing the trigger $\langle attr, \bullet\!\!\twoheadrightarrow, a : b \rangle$ introduces a new fact, $a[attr \twoheadrightarrow c]$, and the resulting H-structure is no longer a model, as it violates the last clause. Therefore, to obtain a model that accommodates the inherited fact, we have to apply the last rule, which derives $a[attr \twoheadrightarrow d]$. Note that derivations that are performed after trigger-firing may affect both the recipient object of the trigger (as shown above) and the source-class (as was shown earlier, in (23)).

The above ideas can be captured via the notion of *one-step inheritance*, defined below. In this paper we limit our attention to the case of Horn F-programs. An extension of this semantics to include programs with negation is reported in [49].

Definition 14.2 (One Step Inheritance Transformation) Let $\mathbf{P}$ be a Horn F-program, $\mathbf{I}$ and $\mathbf{J}$ be H-models of $\mathbf{P}$, and τ be a trigger in $\mathbf{I}$. We say that $\mathbf{J}$ is obtained from $\mathbf{I}$ via τ by *one step of inheritance*, written $TI^{\tau}_{\mathbf{P}}(\mathbf{I}) = \mathbf{J}$, if and only if $\mathbf{J}$ is the unique minimal model among those models of $\mathbf{P}$ that contain $I^{\tau}_{\mathbf{P}}(\mathbf{I})$.

When the actual trigger, τ, is immaterial, we shall write $\mathbf{J} \ll^{inh}_{1} \mathbf{I}$, and $\ll^{inh}$ will denote the reflexive and transitive closure of $\ll^{inh}_{1}$. Informally, $\mathbf{J} \ll^{inh} \mathbf{I}$ means that when inheritance is taken into account, $\mathbf{J}$ is preferable to $\mathbf{I}$ because it has more inherited facts. □

[25] Since $\mathbf{I}(cl[m@a_1, ..., a_n], \twoheadrightarrow)$ may contain $\emptyset$, v may denote this element. In this case, $obj[m@a_1, ..., a_n \twoheadrightarrow v]$ should be understood as $obj[m@a_1, ..., a_n \twoheadrightarrow \{\}]$.

$\bullet\!\!\twoheadrightarrow$) as follows:

$$
\begin{array}{lcl}
\mathbf{I}(p[m@a_1,...,a_n],\rightarrow) & = & \{v \mid p[m@a_1,...,a_n \rightarrow v] \in \mathbf{I}, v \in U(\mathcal{F})\} \\
\mathbf{I}(p[m@a_1,...,a_n],\ \bullet\!\!\rightarrow) & = & \{v \mid p[m@a_1,...,a_n \bullet\!\!\rightarrow v] \in \mathbf{I}, v \in U(\mathcal{F})\} \\
\mathbf{I}(p[m@a_1,...,a_n],\twoheadrightarrow) & = & \{v \mid p[m@a_1,...,a_n \twoheadrightarrow v] \in \mathbf{I}, v \in U(\mathcal{F})\} \\
 & & \cup\ \{\emptyset \mid \text{if } p[m@a_1,...,a_n \twoheadrightarrow \{\,\}] \in \mathbf{I}\,\} \\
\mathbf{I}(p[m@a_1,...,a_n],\ \bullet\!\!\twoheadrightarrow) & = & \{v \mid p[m@a_1,...,a_n \bullet\!\!\twoheadrightarrow v] \in \mathbf{I}, v \in U(\mathcal{F})\} \\
 & & \cup\ \{\emptyset \mid \text{if } p[m@a_1,...,a_n \bullet\!\!\twoheadrightarrow \{\,\}] \in \mathbf{I}\}
\end{array}
$$

Observe that $\mathbf{I}(p[m@a_1,...,a_n],\twoheadrightarrow)$ is either empty, when m is undefined on p as a set-valued method (invoked with the arguments $a_1,...,a_n$); or it is a heterogeneous set that contains $\emptyset$ (the empty set) as an element and, possibly, some elements from $U(\mathcal{F})$. Similarly, when $\mathbf{I}(p[m@a_1,...,a_n],\rightarrow)$ is empty, m is undefined on p, when it is interpreted as a scalar method. Otherwise, if m is defined, then $\mathbf{I}(p[m@a_1,...,a_n],\rightarrow)$ is a homogeneous set of elements taken from the Herbrand universe, $U(\mathcal{F})$. Note that this set may contain more than one element (even though the invocation is scalar) because of the effects of the equality operator: if $t \in \mathbf{I}(p[m@a_1,...,a_n],\rightarrow)$ and $(t\doteq s) \in \mathbf{I}$, then also $s \in \mathbf{I}(p[m@a_1,...,a_n],\rightarrow)$. Similar observations apply to $\mathbf{I}(p[m@a_1,...,a_n],\ \bullet\!\!\rightarrow)$ and $\mathbf{I}(p[m@a_1,...,a_n],\ \bullet\!\!\twoheadrightarrow)$.

Definition 14.1 (Inheritance Triggers) Let $\mathbf{I}$ be an H-structure. Consider a tuple $\tau = \langle m, \bullet\!\!\rightarrow, obj\sharp cl, a_1, ..., a_n\rangle$, where $obj, cl, m, a_1, ..., a_n \in U(\mathcal{F})$ are ground id-terms, and $\sharp$ denotes the type of the is-a relation (":" or "::") that exists between obj and cl. We shall say that τ is an *active inheritance trigger* in $\mathbf{I}$ (or, just a *trigger*, for brevity) if and only if the following conditions hold:

- $obj\sharp cl \in \mathbf{I}$,[24] and there is no intervening class, $mid \in U(\mathcal{F})$, such that $mid \neq cl$ and $obj\sharp mid, mid :: cl \in \mathbf{I}$;
- The method m is *defined* in $\mathbf{I}$ as an inheritable property of cl with arguments $a_1, ..., a_n$; *i.e.*, $\mathbf{I}(cl[m@a_1,...,a_n],\ \bullet\!\!\rightarrow) \neq \emptyset$; and
- m is *undefined* in $\mathbf{I}$ on obj with arguments $a_1, ..., a_n$. More precisely, if "$\sharp$" is ":" (*i.e.*, if obj is a member of cl) then it must be the case that $\mathbf{I}(obj[m@a_1,...,a_n],\rightarrow) = \emptyset$. Otherwise, if "$\sharp$" is a subclass relationship, "::", then $\mathbf{I}(obj[m@a_1,...,a_n],\ \bullet\!\!\rightarrow) = \emptyset$.

Triggers are defined for set-valued invocations of methods similarly, by replacing $\rightarrow$ with $\twoheadrightarrow$ and $\bullet\!\!\rightarrow$ with $\bullet\!\!\twoheadrightarrow$. □

It is clear from the definition that an inheritance trigger is like a loaded gun waiting to fire. Firing a trigger leads to a derivation by inheritance of new facts that cannot be derived using classical deduction. After firing, the trigger is "deactivated" and, in fact, is no longer a trigger in the resulting H-structure.

Given a trigger of the above form, we will say that obj is a *recipient* object of the impending inheritance step and cl is the *source class*. Note that obj here may represent an individual object as well as a subclass.

[24] That is, either $obj : cl$ or $obj :: cl$ is in $\mathbf{I}$, depending on the specific relationship denoted by "$\sharp$".

Our choice is to not undo inheritance in such a case and, at the same time, to not inherit the newly derived fact, $b[attr \bullet\!\!\twoheadrightarrow d]$. The reason for the former is primarily computational, as this leads to much less backtracking. The reason for the latter is aesthetic, as it leads to a more uniform definition. Another way of looking at this is as follows: once the initial inheritance has taken place, the attribute *attr* is already defined on a and so by the time $b[attr \bullet\!\!\twoheadrightarrow d]$ is derived the inheritance of $attr \bullet\!\!\twoheadrightarrow d$ by a is overwritten.

To illustrate the problems that (iii) may cause, consider the following program:

$$\begin{array}{ll} a : p & p[attr \bullet\!\!\rightarrow c] \\ a : t \longleftarrow a[attr \rightarrow c] & t[attr \bullet\!\!\rightarrow d] \\ t :: p \longleftarrow a[attr \rightarrow c] & \end{array} \tag{24}$$

In this example, if a inherits $attr \rightarrow c$ from p the two deductive rules in the program are activated and then both, $a : t$ and $t :: p$, are derived. In other words, t is propelled into the middle of the inheritance path from p to a. What is unusual here is the fact that the attribute *attr* is already defined on t and has a value, d, which is different from c. This means that, had t existed on the inheritance path right from the start, the inheritance of $attr \bullet\!\!\rightarrow c$ from p to a would have been written over, and a would have inherited $attr \bullet\!\!\rightarrow d$ instead. However, in (24), t was *not* on the inheritance path initially and so the above argument does not apply. Nevertheless, one can argue that the subsequent derivation of $a : t$ and $t :: p$ undermines the basis for the above inheritance step. Thus, the question is, should such an inheritance step be undone?

Again, our choice is to not undo such steps. One important reason is, as before, the efficiency of the computation. The other—no less important—reason is that this leads to a very simple semantics whose implications are easy to follow.

In a nutshell, the idea is to decompose each inheritance step into a pair of sub-steps: 1) a "pure" inheritance step, which may introduce new facts but whose output H-structure may no longer be a model of the original program; and 2) a derivation step that turns the result of the previous step into a model. These operations are repeated until inheritance can be done no more.

In this connection, we should mention the recent work [34], which also proposes a semantics for inheritance in a language derived from F-logic. In particular, this work addresses the issue of the interaction between inheritance and deduction. However, it does not account for set-valued methods and for dynamically changing IS-A hierarchies, which are two of the three difficult issues listed above. This approach is also fundamentally different from ours in the way semantics is defined. What [34] calls a "canonic model" of a program is actually not a model of that program in the usual sense. Instead, rules that are found to be written over do not have to be satisfied. This aspect of the semantics in [34] is analogous to the work on ordered logic [61, 62], discussed earlier. In contrast, our semantics is more traditional and, in particular, our "canonic models" are also models in the usual sense.

14.2.2 A Fixpoint Semantics for Non-monotonic Inheritance

Let **I** be an H-structure and $p, m, a_1, ..., a_n$ be ground id-terms. We define the result of application of a method, m, on the object p with arguments $a_1, \ldots, a_n$ and for a given invocation type ($\rightarrow$, $\twoheadrightarrow$, $\bullet\!\!\rightarrow$, or

each canonical model as a viable "possible world" and so any one of these models can be chosen non-deterministically; the other interpretation is that only the facts that belong to the intersection of all such models can be trusted. Since our primary concern is programming, we adopt the former view. In other words, when inheritance conflict occurs due to multiple inheritance, any one of the canonic models can be selected non-deterministically and then used to answer queries.

In a different setting, non-monotonic inheritance was also discussed in [61, 62, 21]. According to these approaches, a logical object (or a class) is a set of rules that represent "knowledge" embodied by the object (resp., class). These rules can be inherited by objects that are placed lower in the hierarchy. Inheritance can be overwritten based on various criteria (*e.g.*, the existence of overriding rules, as in [61, 62], or because of higher-priority rules, as in [21]).

Apart from the very different setup in which behavioral inheritance takes place in these works, there is another fundamental difference with our approach (and, for that matter, with [36, 91] and related works). Namely, in [61, 62, 21], what is inherited is a set of program *clauses*, while in F-logic it is ground data expressions that are passed down the IS-A hierarchy. The latter approach seems to be more flexible. As shown in Section 14.3, F-logic can easily account for inheritance of clauses and for some other forms of inheritance (see Section 14.3), whose representation in the framework of [61, 62, 21] is not obvious to us.

14.2.1 Informal Introduction to the Approach

To integrate inheritance into an object-oriented logic programming system such as F-logic, the following issues must be addressed:

(i) The interaction between inheritance and ordinary logical deduction;

(ii) Inheritance of sets; and

(iii) The dynamic nature of the IS-A hierarchy.

These issues appear to be closely related and, in fact, it is (ii) and (iii) that makes (i) a hard problem. The difficulty is that after inheritance is done, a program clause may become "active," causing other facts to be derived. This new inheritance may affect the recipient object and, even more curiously, the source-class of inheritance. To illustrate, consider the following program:

$$\begin{array}{l} a : b \\ b[attr \bullet\!\!\rightarrow\!\!\!\rightarrow c] \\ b[attr \bullet\!\!\rightarrow\!\!\!\rightarrow d] \Longleftarrow a[attr \rightarrow\!\!\!\rightarrow c] \end{array} \tag{23}$$

If a inherits $attr \bullet\!\!\rightarrow\!\!\!\rightarrow c$ from b, the atom $a[attr \rightarrow\!\!\!\rightarrow c]$ is derived and the last clause is activated. This causes the derivation of $b[attr \bullet\!\!\rightarrow\!\!\!\rightarrow d]$. The question now is, should the inheritance be "undone" in such a case (because—it can be reasoned—the object a should have inherited $attr \bullet\!\!\rightarrow\!\!\!\rightarrow \{c, d\}$ in its entirety or nothing at all)? It seems that different decisions are possible here, and in some cases the choice may be the matter of taste.

monly known as Nixon's Diamond:

$$\begin{array}{ll} nixon : quaker & quaker[policy \bullet\!\!\rightarrow pacifist] \\ nixon : republican & republican[policy \bullet\!\!\rightarrow hawk] \end{array} \tag{21}$$

As a member of two classes, *quaker* and *republican*, *nixon* can inherit either $policy \bullet\!\!\rightarrow pacifist$ or $policy \bullet\!\!\rightarrow hawk$. However, if one of these properties, say $policy \bullet\!\!\rightarrow pacifist$, is inherited by *nixon* (and becomes a non-inheritable property, $policy \rightarrow pacifist$), then the other property can no longer be inherited because, in the new state, the attribute *policy* is defined on the object *nixon*, and this definition overrides any further inheritance. Thus, in this case, inheritance leads to an indeterminate result: the derivation of $nixon[policy \rightarrow pacifist]$ or $nixon[policy \rightarrow pacifist]$, depending on which inheritance step is chosen first.

Overriding and multiple inheritance may each cause non-monotonic behavior. A relation of logical entailment, $\mid\!\approx$, is called non-monotonic if for some $\mathbf{P}$, ψ, and ϕ, it is possible that $\mathbf{P} \mid\!\approx \phi$ and $\mathbf{P} \wedge \psi \not\mid\!\approx \phi$. Classical logic, on the other hand, is monotone, since $\mathbf{P} \models \phi$ implies $\mathbf{P} \wedge \psi \models \phi$. Semantics for non-monotonic logics are usually much more involved, and the corresponding proof theories are rarely complete.

To see why overriding may cause non-monotonic behavior, consider the example in (20), and let $\mid\!\approx$ be the logical entailment relation that does "the right thing" for in this situation. Let $\mathbf{P}$ denote the program in (20). Previously, we have argued that the following is the intended inference:

$$\mathbf{P} \quad \mid\!\approx \quad clyde[color \rightarrow white] \tag{22}$$

Now, suppose we add $\psi = clyde[color \rightarrow black]$ to our set of assertions. Because of the overriding, the property $color \rightarrow white$ is no longer inferred by *clyde* and, thus, $\mathbf{P} \wedge \psi \not\mid\!\approx clyde[color \rightarrow white]$.

Multiple inheritance causes non-monotonic behavior for similar reasons. Consider a part of the Nixon's Diamond (21), where it is not yet known that Nixon is a Quaker. In that case, *nixon* would inherit $policy \bullet\!\!\rightarrow hawk$ from the only class, *republican*, where it belongs. However, in a full-blown Nixon's Diamond, one where $nixon : quaker$ is known to hold, $nixon[policy \rightarrow hawk]$ is no longer a certainty, but rather just one of two possibilities.

Nonmonotonic inheritance has been a subject of intensive research (see, *e.g.*, [36, 91, 92, 43, 22, 90, 57, 38, 56, 80]). The main difference between our approach and the above works is that we are developing an inheritance theory for a general-purpose object-oriented language. By contrast, most of the aforesaid papers tend to study inheritance from a very general, philosophical standpoint. This difference in the orientation clearly shows in the languages they use. First, these languages are mostly propositional and, importantly, do not distinguish between properties (*i.e.*, attributes, in our terminology) and classes. Second, these languages are not part of a more general logic programming system, and so they do not raise the difficult issue of the interaction between inheritance and the ordinary logical deduction.

The semantics for inheritance that will be developed in this section is of the *credulous* breed [43], which means that in the presence of an inheritance conflict we are willing to accept multiple canonical models. This phenomenon can be interpreted in two different ways. One way to look at this is to view

Here *clyde* is a member of the class $royalElephant$ and so it inherits $color \bullet\!\!\rightarrow white$ from that class (which turns into a non-inheritable data expression, $color \rightarrow white$). In principle, it could have inherited $color \bullet\!\!\rightarrow grey$ from the class *elephant*. However, a more specific class of *clyde*, $royalElephant$, has an overriding inheritable property, $color \bullet\!\!\rightarrow white$. Likewise, $royalElephant$ does not inherit $color \bullet\!\!\rightarrow grey$ from *elephant* because of the aforesaid overriding property. On the other hand, $royalElephant$ inherits $group \bullet\!\!\rightarrow mammal$ as an inheritable property, and *clyde* inherits it as a non-inheritable property, $group \rightarrow mammal$.

Note that (20) uses inheritable expressions to indicate that inheritance must take place. We remind from Section 3 that inheritable expressions are used in the following way: when such an expression is inherited by a class-member, say *clyde*, it becomes a non-inheritable expression within that member, even if the member also plays the role of a class in some other context. In contrast, when an inheritable expression, *e.g.* $group \bullet\!\!\rightarrow mammal$, is inherited by a class, *e.g.* $royalElephant$, via a subclass relationship it remains an inheritable expression in that subclass, and so it can be passed down to subclasses and members of that class (cf. the inheritance by $royalElephant$ and by *clyde*).

To illustrate the rationale behind the dichotomy of inheritable/non-inheritable expressions, we shall expand the example in Section 3, where *bob* was represented as a member of the faculty.

Suppose that, for one reason or another, the information about *bob*'s high school years and his years at Yale needs to be scrutinized. One way to represent this information is to create objects *bobInHighschool* and *bobAtYale* and make them completely unrelated to the already existing object *bob*. The disadvantage of this approach is that then we will have to duplicate *bob*'s date of birth, gender, etc. A better way is to turn the new objects into members of class *bob* as follows:

$$
\begin{array}{l}
bob : faculty \\
bobInHighschool : bob[graduation \rightarrow 1968] \\
bobAtYale : bob[graduation \rightarrow 1972] \\
faculty[highestDegree \bullet\!\!\rightarrow phd] \\
bob[birthdate \bullet\!\!\rightarrow 1950; gender \bullet\!\!\rightarrow \text{“}male\text{”}; address \rightarrow \text{“}NewYork\text{”}]
\end{array}
$$

Here *bob* is a member of class *faculty*, while *bobInHighschool* and *bobAtYale* are members of *bob*. Additionally, the second and the third clauses state that *bob* graduated from high school in 1968 and from Yale in 1972. Now, being a member of *faculty*, *bob* inherits $highestDegree \bullet\!\!\rightarrow phd$, yielding a new fact, $bob[highestDegree \rightarrow phd]$. Notice that the property $highestDegree \bullet\!\!\rightarrow phd$ inherited from *faculty* turns into a non-inheritable property of *bob*, since inheritable properties are passed down to class members as non-inheritable properties. Because of that, $highestDegree \rightarrow phd$ can no longer be propagated to the members of class *bob*. And, indeed, it makes little sense to attribute a scholarly degree to Bob when he was in high school or a student at Yale. Likewise, the property $address \rightarrow \text{“}NewYork\text{”}$ of *bob* should not be inheritable because, in all likelihood, Bob had a different address while in high school, and certainly a different address while at Yale. On the other hand, *birthdate* and *gender* are specified as inheritable properties, because these characteristics are not likely to change with time.

The phenomenon of multiple inheritance can be illustrated with the following example that is com-

Other useful encapsulation policies can be represented along similar lines. We should mention that the above definition of type correctness has the same drawback as the notion presented in Section 12: being purely semantic, it is weaker than desired. For instance, suppose that, no faculty has a recorded birthdate in the database. Then adding the clause (19) to **faculty** would still yield a well-typed program, until a faculty with a proper birth date is inserted. Following this update, the program would become ill-typed, and in this way the use of the **person**-private method *birthdate* is precluded inside the module **faculty**. It should be clear, however, that this weakness comes from our notion of type correctness—it is not inherent in our treatment of encapsulation.

14 Inheritance

The concept of inheritance is fundamental in AI and in object-oriented programming, and a number of researchers had worked on combining this idea with programming languages. There are two main aspects of inheritance: *structural* inheritance and *behavioral* inheritance. Structural inheritance is a mechanism for propagating method declarations from superclasses to their subclasses. On the other hand, behavioral inheritance propagates what methods actually do rather than how they are declared.

14.1 Structural Inheritance

Structural inheritance is a subject of many works in functional and logic languages. Cardelli [25] considers inheritance in the framework of functional programming. He described a type inference procedure that is sound with respect to the denotational semantics of his system. Sound type inference systems for functional languages were also discussed in [74, 33] and in several other papers.

LOGIN and LIFE [6, 7] incorporate structural inheritance into logic programming via a unification algorithm for ψ-terms—complex structures that are related to signatures in F-logic but are different, both semantically and syntactically.

In contrast to the above works, F-logic is a full-fledged logic in which structural inheritance is built into the semantics, and the proof procedure is *sound* and *complete* with respect to this semantics. Structural inheritance was discussed in Section 7.3.

14.2 Behavioral Inheritance

The main difficulty in dealing with behavioral inheritance is the fact that it is *non-monotonic*, which is mostly due to the property called *overriding* and also because of the phenomenon of *multiple inheritance*.

Overriding means that any explicit definition of a method takes precedence over any definition inherited from a superclass. For instance, consider the following F-program:

$$\begin{array}{ll} royalElephant :: elephant & elephant[color \bullet\!\!\rightarrow \text{“}grey\text{”}; group \bullet\!\!\rightarrow mammal] \\ clyde : royalElephant & royalElephant[color \bullet\!\!\rightarrow \text{“}white\text{”}] \end{array} \qquad (20)$$

(i) both signatures are annotated with the same keyword;[23] and

(ii) if the annotation is **private** or **export-to**, then the rules ρ_1 and ρ_2 belong to one module.

A module structure is *coherent* if it is coherent with respect to all canonic H-models of $\mathbf{P}$. □

Given a program, $\mathbf{P}$, an H-model, $\mathbf{I}$, and a module, $\mathcal{U}$, of $\mathbf{P}$, the *restriction* $\mathbf{I}$ on $\mathcal{U}$, denoted $\mathbf{I}(\mathcal{U})$, is the smallest H-structure that contains the following atoms:

- every data-atom, is-a atom, or a P-atom in $\mathbf{I}$ that occurs in an active rule in $\mathbf{P}^*_{\mathcal{U}}$; and
- every signature atom in $\mathbf{I}$ that is:
  - a publicly annotated signature occurring in the head of an active rule in $\mathbf{P}^*$; or
  - a privately annotated signature that occurs in the head of an active rule in $\mathbf{P}^*_{\mathcal{U}}$; or
  - a signature explicitly exported into $\mathcal{U}$ from another module, $\mathcal{U}'$, and such that it occurs in the head of an active rule in $\mathbf{P}^*_{\mathcal{U}'}$.

In other words, $\mathbf{I}(\mathcal{U})$ takes all data atoms that are relevant to the module and all signature atoms that are either derivable within that module or are exported by other modules. For instance, the restriction on **administrative** of a canonic model of a program that includes the three modules (16) – (18) will contain all data atoms that appear in the active instances of the last four clauses in (18), all the signatures in that module, all signatures from module **faculty**, and all signatures in **person**, except for $birthdate \rightarrow date$.

We can now define the notion of type-correctness under meta-annotations as follows:

Definition 13.3 (Type-Correctness with Meta-Annotations) Let $\mathbf{P}$ be an F-program with a coherent module structure. Then $\mathbf{P}$ is said to be *type-correct* (or *well-typed*) if it is well-typed in the usual sense (with respect to Definition 12.5) and if every module of $\mathbf{P}$ is also well-typed.
A module, $\mathcal{U}$, is *well-typed* if $\mathbf{L}(\mathcal{U})$ is a typed H-model of $\mathbf{P}_{\mathcal{U}}$ for every canonic model, $\mathbf{L}$, of $\mathbf{P}$. □

To illustrate this definition, consider, again, the module structure (16) – (18). This program is well-typed because all signatures used in these modules are properly exported. Suppose now that we add

$$F[birthmonth \rightarrow M] \longleftarrow F : faculty\,[birthdate \rightarrow d(D, M, Y)] \tag{19}$$

to module **faculty**. The resulting program will not be type-correct, because the module **faculty** will no longer be well-typed.

To see why, let $\mathbf{L}$ be a canonic model of this program. The restriction $\mathbf{L}$(**faculty**) does not contain the signature $faculty[birthdate \Rightarrow date]$, since the latter does not appear in the head of any active rule local to **faculty**, and it is not exported into this module. On the other hand, if there is even a single faculty, say *fred*, for whom birthdate is recorded, then $fred[birthdate \rightarrow \cdots]$ will be in $\mathbf{L}$(**faculty**). Since this data atom is not covered by any signature in $\mathbf{L}$(**faculty**), the latter is not a typed H-structure.

[23] **export-to(a)** and **export-to(b)** are considered to be the same keyword for this purpose.

(ii) An annotation (**private**, **public**, or **export-to**(module)) attached to each signature expression that occurs in the head of a rule in $\mathbf{P}$. □

Note that since modules do not necessarily consist of disjoint sets of clauses, one module may be a sub-module of another module. Note further that, as illustrated above, some modules may represent authorization domains for various classes of users. Clauses and queries introduced by these users are assumed to enter the appropriate modules automatically. Also, Definition 13.1 does not assume that there is a correspondence between modules and classes; however, if desired, modules can be made to fit exactly one class, for any class in a given finite set. Finally, it is important to keep in mind that module structures do not affect the canonic model semantics of the program—only the notion of type correctness is affected, as explained below.

The above definition of module structures is very general and calls for further specialization. One thing that immediately comes to mind is that certain module structures may have internal contradictions. For instance, if a method is declared as public in one module and as private in another, a problem may arise if these declarations clash. We consider a module structure with this property as *incoherent.*

Another situation where a module structure may be viewed as incoherent is when the same method expression is declared as private in two separate modules. If this were allowed, any module could import any signature by declaring it private. This policy, however, cannot control what is being *exported* and whereto, and thus it prevents enforcing any sensible access authorization policy. There are also purely software-engineering arguments against this policy, which has led many object-oriented systems to adopting the idea of controlled export rather than import. A consequence of this policy is that duplication of private declarations in different modules is not allowed. Similarly, we shall assume that a signature can be exported from only one module—the module where it is declared. These ideas are formally captured via conditions (i) and (ii) of Definition 13.2, below.

To formulate the notion of coherence, we need a few simple definitions. Let $\mathcal{L}$ be an F-logic language, $\mathbf{P}$ be a program with a module structure, and let $\mho$ be a module of $\mathbf{P}$. Let $\mathbf{P}_\mho$ denote the set of clauses in $\mathbf{P}$ that belong to $\mho$. The notation $\mathbf{P}^*$ (or $\mathbf{P}^*_\mho$) will be used to denote the set of *annotated* ground instances of $\mathbf{P}$ (resp., $\mathbf{P}_\mho$). We shall assume that rule-instantiation preserves annotations of signature expressions occurring in the heads of the rules in $\mathbf{P}$, and that instances of the signatures that have different annotations are not merged.[21]

Let $\mathbf{I}$ be an H-structure of $\mathbf{P}$. (Note: $\mathbf{I}$ has no annotations, but $\mathbf{P}$ and $\mathbf{P}^*$ have). A ground rule in $\mathbf{P}^*$ is *active* in $\mathbf{I}$ if all its premises are true in $\mathbf{I}$. If $\mathbf{I}$ is a model, then the heads of all active rules are also true in $\mathbf{I}$.

Definition 13.2 (Coherent Module Structures) A module structure for $\mathbf{P}$ is *coherent* with respect to a canonic H-model, $\mathbf{L}$, if for every data-atom in $\mathbf{L}$ that is covered[22] by a pair of signatures that occur in the heads of some active rules, $\rho_1, \rho_2 \in \mathbf{P}^*$ (where $\rho_1 \equiv \rho_2$ is possible), the following holds:

[21] For instance, instantiating X and Y to b in a[**private** : $X \rightarrow c$; **public** : $Y \rightarrow c$] yields a[**private** : $b \rightarrow c$; **public** : $b \rightarrow c$], where the two identical instances of $b \rightarrow c$ are kept separately due to the difference in annotations.

[22] Covering is defined in Definitions 12.1 and 12.2.

The above represents a fragment of the definition of the module **faculty**. The methods *account* and *balance* are not visible by any module except **faculty** and a module where administrative workers process project accounts. In this particular case, the reason for shielding *account* and *balance* is that of protection rather than encapsulation—we assume that the expenditures in project accounts are not in the public domain. However, the total amount of funds initially set aside for the project is made available through the public method *funding*.

$$
\begin{array}{l}
\textbf{module}\ \ \textbf{administrative}\ \{ \\
\qquad budget[\ \ \textbf{public}: \quad salaries \Rightarrow int;\ equipment \Rightarrow int;\ supplies \Rightarrow int;\ total \Rightarrow int\,] \\
\qquad account[\ \ \textbf{private}: \quad project \Rightarrow project; \\
\qquad\qquad\qquad\ \ \textbf{public}: \quad budget \Rightarrow budget;\ expended \Rightarrow int;\ encumbered \Rightarrow int; \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ committed \Rightarrow int;\ balance \Rightarrow int\,] \\
\quad B\,[total \rightarrow X+Y+Z\,] \ \longleftarrow\ \ B : budget\,[salaries \rightarrow X;\ equipment \rightarrow Y;\ supplies \rightarrow Z\,] \\
\quad A\,[committed \rightarrow X+Y\,] \ \longleftarrow\ \ A : account\,[expended \rightarrow X;\ encumbered \rightarrow Y\,] \\
\quad A\,[balance \rightarrow X-Y\,] \ \longleftarrow\ \ A : account\,[budget \rightarrow Z[total \rightarrow X];\ committed \rightarrow Y\,] \\
\quad ?-\ fred[balance@bluff \rightarrow X] \\
\}
\end{array}
\tag{18}
$$

Unlike module **person**, the last two modules, **administrative** and **faculty**, are not built around any specific class. Apart from being depositories for method definitions, these modules serve as authorization domains for the members of the faculty and for workers who administer project accounts. Note that **faculty** exports two methods to module **administrative**, which makes their invocation in that module legal (cf. the last query in **administrative**). Also, although most of the information about an account is made public in module **administrative**, the identifying information of the account is not, which is achieved by making the attribute *project* in class *account* (project associated with account) private.

In the above example, the construct **module** $\{\ \cdots\ \}$ and the keywords **private**, **public**, and **export-to** are not part of the logic but, rather, are *meta-annotations*. Their purpose is to affect the notion of type correctness presented in Section 12, which itself is a meta-logical concept. The following section gives a formal account of type-correctness for programs with a module structure.

13.2 Modules and Type Correctness

The above example contains all major elements of our treatment of encapsulation. The key idea is to use a stronger notion of type-correctness to preclude illegal uses of methods with limited scope. This requires imposition of a *module structure* on each F-program, which is specified via meta-logical annotations. The new notion of type correctness builds on the notion introduced in Section 12 and reduces to the old notion when the entire F-program is considered as one big module.

Definition 13.1 (Module Structures) Let **P** be an F-program. A *module structure* of **P** consists of the following:

(i) A decomposition of the set of clauses in **P** into a collection of named (not necessarily disjoint) subsets, called *modules*;

First, all program clauses must be grouped into *modules*. Then each module is checked for type-correctness separately. The notion of type-correctness referred to here is more elaborate than the one in Section 12. The main difference is that the set of signatures used in determining type-correctness of any given module now consists not only of signatures determined by the clauses that belong to that module but also of signatures that are exported to this module by other modules.[20]

It should be noted that in most object-oriented systems, encapsulation policies center around the concept of a class, not module. However, in a language where classes can be virtual, *i.e.*, defined via logical clauses, encapsulating each class seems to be infeasible. Nevertheless, modules can be made to fit exactly one class and, thus, they provide a way to encapsulate any *fixed* number of classes (which is all what current systems are capable of doing, anyway).

Traditionally, object-oriented systems were employing the idea of encapsulation for hiding implementation details, and our approach is not different in this respect. However, a combination of modules with type-correctness can serve another useful purpose: authorization-based control over access to objects. This can be achieved along the following lines. First, each user is assigned to a module. Then, each clause or a query that a user adds to the system is automatically made part of that user's module. This simple schema can control unauthorized access, because any user query would constitute a type error (and would be rejected) if it were to invoke a method that is not exported into that user's module.

13.1 An Example

To illustrate the above ideas, consider a program consisting of the following three modules:

$$
\begin{array}{l}
\textbf{module}\ \ \textbf{person}\ \{ \\
\qquad person[\ \ \textbf{private}: \ \ birthdate \Rightarrow date; \\
\qquad\qquad\qquad\ \ \textbf{public}: \ \ age \Rightarrow int;\ name \Rightarrow string;\ \cdots] \\
\qquad P\,[age \rightarrow Y' - Y\,] \ \longleftarrow\ \ P : person[birthdate \rightarrow d(D,M,Y)\,] \wedge today() = d(D',M',Y') \qquad (16) \\
\qquad\quad \%\ \ \text{Assume that dates (the members of class } date\text{) have the form} \\
\qquad\quad \%\ \ d(Day, Month, Year).\ \text{The function } today() \text{ returns today's date.} \\
\}
\end{array}
$$

In this module, the attribute *birthdate* is declared as private in order to encapsulate the implementation of the method *age* from future changes to the structure of the class *person.* For instance, we may replace *birthdate* by *yearOfBirth*, or we may change the type of *birthdate* without impacting methods defined outside the module **person**.

$$
\begin{array}{l}
\textbf{module}\ \ \textbf{faculty}\ \{ \\
\qquad faculty[\ \ \textbf{public}: \ \ funding@project \Rightarrow int;\ projects \twoheadrightarrow project;\ \cdots] \\
\qquad project[\ \ \textbf{export-to(administrative):} \\
\qquad\qquad\qquad\qquad account \Rightarrow account;\ balance@project \Rightarrow int;\ \cdots] \\
\qquad F\,[funding@P \rightarrow X] \ \longleftarrow\ \ F : faculty\,[projects \twoheadrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad P\,[account \rightarrow A[budget \rightarrow B[total \rightarrow X]\,]\,]\,] \qquad (17) \\
\qquad F\,[balance@P \rightarrow X\,] \ \longleftarrow\ \ F : faculty\,[projects \rightarrow P[account \rightarrow A[balance \rightarrow X]\,]\,] \\
\}
\end{array}
$$

[20] Public signatures are considered to be exported to every module.

with restricted classes of F-programs, *e.g.*, [42]). On the other hand, while Definition 12.5 accommodates a vast class of programs, it is weaker than we would like it to be. Essentially, our notion of type-correctness amounts to saying that well-typed programs are not allowed to derive facts that are inconsistent with signatures specified for these programs. Clearly, this is a minimum that must be required of a correct logic program. However, another source of type errors comes from rules that have signature-incompatible body-literals. Unfortunately, this latter kind of errors is not captured by Definition 12.5. A detailed discussion of these issues and some solutions can be found in [98, 29].

The semantics of type-correctness presented here is of the *type-checking* variety, which means that it requires the programmer to supply a *complete* type specification for each program. However, in logic programming, supplying complete type specifications is often viewed as a bother. Adapting the ideas from ML to Prolog, Mishra [73] proposed to use *type-inference* for determining predicate types and then use the inferred types for trouble-shooting logic programs.

Semantics of type inference is different from the semantics of type checking. Briefly, given a program, **P**, its inferred type is the "strongest" type declaration under which **P** is well-typed. If pure type-checking is viewed by some as too taxing, pure type-inference appears to be at another extreme and so it is used mostly as an advisory tool. We feel—as suggested by other researchers in the past [101, 99]—that type-inference combined with type checking is a suitable compromise. A way to define logical semantics for this approach was proposed in [51, 98].

13 Encapsulation

Encapsulation is a major concept in the suite of notions comprising the object-oriented paradigm. In a nutshell, encapsulation is a software-engineering technique that requires special designations for each method in a class. Methods designated as *public* can be used to define other methods in other classes. In contrast, methods designated as *private* can be used only in the class where they are declared. Other methods may be exported to some specific classes, but not to all classes.

While the idea of encapsulation is simple, its logical rendition is not. In [71], Miller proposes a way to represent modules in logic programming via the intuitionistic embedded implication. Chen [28] defines modules as second-order objects, where data encapsulation inside modules is represented using existential quantifiers. In their present form, these methods are not sufficiently general to be applied to F-logic. It would be interesting to see, though, if these approaches can be extended to make them suitable for method encapsulation in an object-oriented logic.

In contrast to [71, 28], we view various encapsulation mechanisms simply as elaborate type-correctness policies. This approach is quite general and is capable of modeling a wide variety of encapsulation mechanisms. Its other advantage is that it does not require extension to the logic. Instead, encapsulation is treated as a type-correctness discipline and, thus, it is a *meta-logical* notion in our approach.[19]

The general outline of our encapsulation-as-type-correctness approach can be formulated as follows.

[19] An interesting feature of [71, 28] is that they achieve encapsulation by purely logical means.

This model satisfies the well-typing conditions of Definition 12.3 and, therefore, is a typed F-structure. Hence, **P** is well-typed.

Suppose now that **P** contains the term $mary[boss \rightarrow john]$ instead of $mary[boss \rightarrow bob]$. Then **P** would have had a type error because the atom $mary[boss \rightarrow john]$ clashes with the typing $empl[boss \Rightarrow faculty]$. Indeed, $john$ is not known to be a faculty and hence $\neg john : faculty$ holds in every canonic model of **P**. Likewise, the clause $mary[salary \rightarrow 10000]$ would have caused a type error. This happens because no signature was specified for the 0-ary version of the method $salary$ (only the unary version of $salary$ has a signature in **P**), and so $\neg faculty[salary \Rightarrow ()]$ and $\neg empl[salary \Rightarrow ()]$ must hold in the minimal model of **P**. But this, together with $mary[salary \rightarrow 10000]$, defeats the first well-typing condition of Definition 12.3.

For another example, consider:

$$\begin{array}{l} ibm : company \\ john[worksFor \rightarrow ibm] \\ X : employee \longleftarrow X[worksFor \rightarrow I] \\ X[worksFor \Rightarrow company] \longleftarrow X : employee \end{array} \tag{15}$$

This F-program is well-typed because its minimal model is also a typed H-model. In contrast, if the last rule in (15) were

$$X[name \Rightarrow string] \longleftarrow X : employee$$

the program would not be well-typed because the attribute $worksFor$ would have had no declared signature and, hence, $\neg c[worksFor \Rightarrow ()]$ would hold in that model, for every class c. This, together with $john[worksFor \rightarrow ibm]$, defeats the first well-typing requirement to typed H-structures (Definition 12.1). Therefore, even though the modified program has a minimal model, it is not a typed model and so the program is ill-typed.

Discussion

Definition 12.5 provides semantic grounds for developing static type-checking algorithms. It also supplies a yardstick for verifying correctness of various such algorithms, and thus provides a uniform framework for comparison. Static type checking is a topic for further research; for classical logic programs, a similarly defined notion of type-correctness was studied in [98]. A detailed comparison of our approach with other works on type systems for logic programming can be found in [51, 98].

To better see the role of type correctness in the overall schema of things, observe that this notion is not part of F-logic but, rather, belongs to its meta-theory. As such, neither the semantics, nor the proof theory depends on the particular way well-typing is defined. Even the sublogic of signatures is independent of Definition 12.5. Therefore, other definitions of type-correctness can also be used in conjunction with F-logic.

The need for a new notion of type-correctness for F-logic arose because existing theories are not general enough to adequately type many kinds of F-programs (although some could be adapted for use

that there should be a signature defined for some class that contains *faculty* as a member and that covers the invocation $faculty[avgSalary \rightarrow 50000]$.[18]

Definition 12.2 (Typed H-structures—Inheritable Data Expressions) Let $\mathbf{I}$ be an H-structure. If α is an inheritable data atom of the form $c[m@a_1, ..., a_k \bullet\!\!\rightsquigarrow v] \in \mathbf{I}$ and β is a signature atom of the form $d[m@b_1, ..., b_k \approx\!\!> \cdots]$, where $\bullet\!\!\rightsquigarrow$ and $\approx\!\!>$ stand, respectively, for $\bullet\!\!\rightarrow$ and $\Rightarrow$ or for $\bullet\!\!\twoheadrightarrow$ and $\Rightarrow\!\!>$, we shall say that β *covers* α if $a_i : b_i$, $c :: d \in \mathbf{I}$.

An H-structure, $\mathbf{I}$, is said to be *typed with respect to inheritable* data expressions if the following conditions hold:

- Every inheritable data atom in $\mathbf{I}$ is covered by some signature in $\mathbf{I}$; and
- If an inheritable data atom, $c[m@a_1, ..., a_k \bullet\!\!\rightsquigarrow v] \in \mathbf{I}$, is covered by a signature of the form $d[m@b_1, ..., b_n \approx\!\!> w] \in \mathbf{I}$, where $v, w \in U(\mathcal{F})$ and $\bullet\!\!\rightsquigarrow$ and $\approx\!\!>$ are as before, then $v : w \in \mathbf{I}$. □

Definition 12.3 (Typed H-Structures—All Data Expressions) The four conditions in Definitions 12.1 and 12.2 are called the *well-typing* conditions. An H-structure, $\mathbf{I}$, is *typed* if all well-typing conditions are satisfied, *i.e.*, if $\mathbf{I}$ is typed with respect to inheritable and non-inheritable data expressions. □

We are now ready to define the notion of well-typed F-programs:

Definition 12.4 (Typed Canonic Models) A *typed canonic model* of $\mathbf{P}$ is a (usual) canonic model for $\mathbf{P}$ that, in addition, is a typed H-structure. □

Definition 12.5 (Well-typed Programs) An F-program $\mathbf{P}$ is *well-typed* (or *type-correct*) if every canonic H-model of $\mathbf{P}$ is a typed canonic H-model. Otherwise, $\mathbf{P}$ is said to be *ill-typed*. □

This definition appears to capture the intuition about type errors quite adequately: If $\mathbf{P}$ has a non-typed canonic model, the problem can clearly be traced to typing. For example, consider the following F-program, $\mathbf{P}$:

$$
\begin{array}{lll}
bob : faculty & 1989 : year & empl[boss \Rightarrow faculty;\ salary@year \Rightarrow integer] \\
mary : faculty & 10000 : int & mary[boss \rightarrow bob] \\
faculty :: empl &&
\end{array}
$$

The minimal H-model of $\mathbf{P}$ consists of the following molecules (and their derivatives):

$$
\begin{array}{lll}
faculty :: empl & bob : faculty & 1989 : year \\
bob : empl & mary : faculty & 10000 : int \\
mary : empl & mary[boss \rightarrow bob] &
\end{array}
$$
$$
empl[boss \Rightarrow (faculty, empl);\ salary@year \Rightarrow integer]
$$
$$
faculty[boss \Rightarrow (faculty, empl);\ salary@year \Rightarrow integer]
$$

[18] Such a class, called *employmentGroup* was introduced towards the end of Section 3.

attributes that are defined for the particular relation at hand, and any attempt to use an attribute that is not defined for this relation will normally result in an error.

To illustrate, we reproduce an example from Section 11.3:

$$empl[salary@year \Rightarrow integer]$$
$$person[birthdate \Rightarrow year]$$

Here we would like to make it illegal for an *empl*-object to call methods other than *birthdate* and *salary*. Similarly, for persons that are not employees, we would like to prohibit the use of the attribute *salary* altogether, since it is not covered by an appropriate signature in class *person*. In other words, we would like to say that (in the above example) the signature of *person*-objects consists precisely of one method, *birthdate*, and that the signature of *empl*-objects has precisely two methods, *salary* and *birthdate*. Both methods are scalar; *birthdate* expects no arguments, while salary needs one—a *year*-object.

Unfortunately, by itself these declarations *do not* ensure that the corresponding typing constraint hold in a canonic H-model, because connection is missing between signatures and data expressions. This missing link is now provided in the form of the well-typing conditions:

Definition 12.1 (Typed H-structures—Non-inheritable Data Expressions) Let $\mathbf{I}$ be an H-structure. If α is a non-inheritable data atom of the form $o[m@a_1, ..., a_k \rightsquigarrow v] \in \mathbf{I}$ and β is a signature atom of the form $c[m@b_1, ..., b_k \twoheadrightarrow \cdots]$, we shall say that β *covers* α if, for each $i = 1, ..., k$, we have $o : c$, $a_i : b_i \in \mathbf{I}$. Here $\rightsquigarrow$ and $\twoheadrightarrow$ stand, respectively, for $\rightarrow$ and $\Rightarrow$ or for $\twoheadrightarrow$ and $\Rightarrow\!\!\!\!\Rightarrow$.

We shall say that $\mathbf{I}$ is a *typed* H-structure *with respect to non-inheritable* data expressions if the following conditions hold:

- Every non-inheritable data atom in $\mathbf{I}$ is covered by a signature atom in $\mathbf{I}$; and
- If a data atom, $o[m@a_1, ..., a_k \rightsquigarrow v] \in \mathbf{I}$, is covered by a signature of the form $c[m@b_1, ..., b_n \twoheadrightarrow w] \in \mathbf{I}$, where $\rightsquigarrow$ and $\twoheadrightarrow$ are as before and $v, w \in U(\mathcal{F})$, then $v : w \in \mathbf{I}$. □

Note that the first condition imposes restrictions on the domain of the definition of methods, and on when a method can be invoked as a scalar or a set-valued method. In plain terms, it says that every non-inheritable expression in $\mathbf{I}$ must be covered by a signature. The second condition says that every non-inheritable expression must satisfy all constraints imposed by the signatures that cover that expression.

When it comes to typing, inheritable expressions are *not* completely analogous to non-inheritable expressions, despite their semantic and proof-theoretic similarities. The reason for this is that inheritable expressions are *used* differently. Indeed, consider the expression $highestDegree \bullet\!\!\rightarrow phd$ of class *faculty* in Section 3. This expression is supposed to be inherited by the members of class *faculty* and thus it is also a property of these members. Therefore, it must obey all typing constraints imposed on *faculty*.

It is instructive to compare the above expression to $avgSalary \rightarrow 50000$, which is a non-inheritable property of *faculty*. Since this expression does not apply to the members of the class, it does not have to comply with the signatures given for the class *faculty*. Instead, Definition 12.1 applies here, which means

This part of the job is quite standard and can be found in many guides to Prolog. Once the expression is parsed, it has to be evaluated. Surprisingly, evaluators for joins, Cartesian products, and the like are rather cumbersome and take anywhere from one to two pages of Prolog code. The main difficulty here is the fact that arities of the expressions to be joined are unknown at compile time, which calls for the use of various nonlogical features such as "=..", "arg", and "functor". We shall see that a page-plus-long Prolog program for the relational join evaluator can be written in just three rules in F-logic.

Relations are represented as in Section 6, that is, each relation, p, is a class of tuple-objects. Given a pair of relations, p and q, and a join condition, $cond$, the program creates a new relation, $join(p, q, cond)$. Join-conditions are represented as lists of pairs of the form $cond(A, B, cond(\ldots))$. For instance, $cond(A_1, B_1, cond(A_2, B_2, nil))$ encodes an equi-join condition $A_1 = B_1 \wedge A_2 = B_2$. The first deductive rule, below, says that if i and j are oid's of tuples in relations p and q, respectively, then $new(i, j)$ is the oid of a tuple in a nil-join of p and q (*i.e.*, in a Cartesian product of p and q). The second rule recursively computes the oid's of all tuple-objects in the equi-join. These tuple-objects are not fully defined, however, as we have not yet specified the values of their attributes. The third rule, therefore is needed to extract these values from the source-tuples i and j:

$$\begin{aligned} new(I,J) : join(P,Q,nil) \longleftarrow \quad & I : P \wedge\ J : Q \\ new(I,J) : join(P,Q,cond(L,M,Rest)) \longleftarrow \quad & I : P[L \rightarrow X] \wedge J : Q[M \rightarrow X] \\ & \wedge\, new(I,J) : join(P,Q,Rest) \\ new(I,J)[L \rightarrow X, rename(M) \rightarrow Y] \longleftarrow \quad & I : P[L \rightarrow X] \wedge J : Q[M \rightarrow Y] \end{aligned} \tag{14}$$

Note that in the last rule, the attributes coming from the second relation in the join are renamed in order to avoid name clashes. When relations do not have common attributes or when we are interested in natural joins rather than equi-joins, this renaming would not be needed and so the symbol *rename* in the last rule could be dropped. Additionally, for natural joins, the join-condition itself can be simplified to be a list of attributes instead of pairs of attributes.

Now, if *dept* has attributes *dname* and *mngr*, while the relation *empl* has attributes *dname* and *ename*, the query $?- \quad X : join(dept, empl, cond(dname, dname, nil))[Y \rightarrow Z]$ will return a join of the two relations on the attribute *dname*.

Actually, the above program does a more general job since it joins *classes* rather than just relations. It also demonstrates a difference that syntax can make: In Prolog, a purely logical specification of a join entails a rather sophisticated encoding of relations, which—in the end—results in a code that can hardly be called "declarative."

12 Well Typed Programs and Type Errors

In a strongly typed language, a method can be invoked only if this invocation is covered by one of the signatures specified for this method.[17] For instance, in the relational model, one can use only those

[17] By an *invocation* we mean a tuple of the form $\langle obj, \vec{args} \rangle$, where obj is the host-object of the invocation and $\vec{args}$ is a list of arguments of the invocation.

of a general situation where joint projects, common hobbies, and other commonalities may be involved. We can abstract this concept and define a "generic" method for joint things:

$$\begin{array}{rcl} X[joint(M)@nil \twoheadrightarrow Z] & \leftarrow & X[M \twoheadrightarrow Z] \\ X[joint(M)@cons(Obj, Rest) \twoheadrightarrow Z] & \leftarrow & Obj[M \twoheadrightarrow Z] \wedge X[joint(M)@Rest \twoheadrightarrow Z] \end{array}$$

The first rule here describes things that X has in common with an empty list of objects, nil. The second rule, then, says that X has object Z in common with each member of the list $cons(Obj, Rest)$ if Obj has Z and Z is a common object that X shares with the rest of the list.

Our second example describes similarities among classes, specified via the *like*-relationship (cf. [41, 11]). For instance, one can say, "The concept of a Whale is *like* the concept of a Fish via habitat" or, "A Pig-Like-Person is *like* a Pig via nose and legs."

Like-similarity can be expressed by means of a ternary predicate, $like$. For instance, in $like(pig\text{-}like, pig, cons(legs, cons(nose, nil)))$, the third argument would list the attributes that relate the first two arguments. We could then define:

$$C_1[Attr \rightarrow P] \leftarrow C_2[Attr \rightarrow P] \wedge like(C_1, C_2, PropList) \wedge member(Attr, PropList)$$

where $member$ is a membership predicate that can be defined by a standard Prolog program.

The above technique works well for specifying similarity via *attributes*, *i.e.*, via 0-ary methods. To represent similarity via methods of arbitrary arities we would need one $like_n$ predicate, for each arity $n \geq 0$, and one rule of the form

$$\begin{array}{rcl} C_1[M@X_1, ..., X_n \rightarrow P] \leftarrow C_2[M@X_1, ..., X_n \rightarrow P] & \wedge & like_n(C_1, C_2, MethdList) \\ & \wedge & member(M, MethdList) \end{array}$$

11.4.4 List Manipulation

The following is an F-logic counterpart of a canonic textbook example:

$$\begin{array}{l} nil[append@L \rightarrow L] \leftarrow L : list(T) \\ cons(X, L)[append@M \rightarrow cons(X, N)] \quad \leftarrow L : list(T)[append@M \rightarrow N] \wedge X : T \end{array} \tag{13}$$

Here L, M, and N are list-objects, while *append* is a method that applies to list-objects; when invoked with a list-object, b, as an argument on a list-object, a, as a host *append* returns a concatenation of a and b. A parametric family of list-classes, $list(T)$, and their signatures were defined earlier, in (10) of Section 11.2 and in (11) of Section 11.3, respectively.

11.4.5 A Relational Algebra Interpreter

Writing an intelligible Prolog interpreter for relational algebra is rather challenging. First, one should parse the relational expression given as input. For example, an expression $((P \times V) \cup W) \bowtie_{cond} (Q - R)$ would be converted into a term of the form

$$join(\ union(prod(P, V), W),\ minus(Q, R),\ cond\ )$$

Typically, database queries are specified with respect to a fixed, known scheme. Experience shows, however, that this assumption is unrealistic and ad hoc schema exploration may often be necessary. This means that the user has to apply intuitive or exploratory search through the structure of the scheme and the database at the same time and even in the same query (cf. [76]). Many user interfaces to commercial databases support browsing to different extent. The purpose of the following examples is to demonstrate that F-logic provides a unifying framework for data *and* schema exploration. Once again, we refer to the example in Section 3.

For each object in class $faculty$, the following pair of rules collects all attributes that have a value in class $person$:

$$\begin{array}{lcl} interestingAttributes(X)\,[attributes \twoheadrightarrow L] & \leftarrow & X : faculty[L \rightarrow Z : person] \\ interestingAttributes(X)\,[attributes \twoheadrightarrow L] & \leftarrow & X : faculty[L \twoheadrightarrow Z : person] \end{array} \qquad (12)$$

For every $faculty$-object, o, these rules define a new object, $interestingAttributes(o)$, with a set-valued attribute $attributes$. The intuitive reading of (12) is: If L is an attribute that is defined on an object, X, and has a $person$-value, Z, then L must belong to the result that $attributes$ has on the object $interestingAttributes(X)$.

Thus, in the example in Section 3, we would obtain: $interestingAttributes(bob) = \{boss\}$ and $interestingAttributes(mary) = \{friends\}$. Deleting the restriction $person$ in (12) would add those attributes that have any value on X, not just a $person$-object. In that case, $interestingAttributes(bob)$ will also contain $name$, age, and $worksFor$, while $interestingAttributes(mary)$ will include $name$, $highestDegree$, and $worksFor$.

Another interesting example is when we need to retrieve all objects that reference some other object directly or indirectly (via subobjects). The method *find*, below, returns the set of all such objects that reference *Stuff*:

$$\begin{array}{l} browser[find@Stuff \twoheadrightarrow X] \leftarrow X[Y \rightarrow Stuff] \\ browser[find@Stuff \twoheadrightarrow X] \leftarrow X[Y \twoheadrightarrow Stuff] \\ browser[find@Stuff \twoheadrightarrow X] \leftarrow X[Y \rightarrow Z] \wedge browser[find@Stuff \twoheadrightarrow Z] \\ browser[find@Stuff \twoheadrightarrow X] \leftarrow X[Y \twoheadrightarrow Z] \wedge browser[find@Stuff \twoheadrightarrow Z] \end{array}$$

For the example in Section 3, the query $?- browser[find@\text{“}CS\text{”} \twoheadrightarrow X]$ would return the set $\{cs_1, cs_2, bob, mary\}$.

11.4.3 Representation of Analogies

Reasoning by analogy is an active field of research (*e.g.*, see a survey in [40]). Apart from the semantic and heuristic issues, finding suitable languages in which to specify analogies is also a challenge. This subsection shows how certain kinds of analogies can be specified in F-logic.

In Section 3, we defined a method that, when applied on a host object of type $person$ with a $person$-argument, returns the set of all joint works of the two persons involved. This method is just one instance

the other hand, students may be entitled to see their own grades. To implement this policy, each student can be given access to the object that represents the achievements of that student. We can then define the method *grade* on *student*-objects as follows:

$$Stud[grade@Crs \rightarrow G] \leftarrow Crs : course[grade@Stud \rightarrow G]$$

In this way, a student, say Mary, can access her grades by posing queries such as $?- mary[grade@vlsi \rightarrow G]$, but she cannot access grades by querying a *course*-object, *e.g.*, $?- vlsi[grade@mary \rightarrow G]$.

The power-set operator. Another interesting problem is to express the power-set operation, an operator that takes an arbitrary set and constructs its power-set. First, we define the class of sets; it consists of objects with oid's of the form $add(a, add(\ldots))$. The intended meaning of a set-object denoted by $add(a, add(b, add(c, nil)))$ is the set $\{a, b, c\}$. The class of sets is defined as follows:

$$\begin{array}{l} nil : set \\ add(X, S) : set \ \leftarrow \ S : set \\ add(X, add(Y, S)) \doteq add(Y, add(X, S)) \\ add(X, add(X, S)) \doteq add(X, S) \end{array}$$

The first equation asserts that the order of adding elements to sets is immaterial; the last equation says that insertion of elements into sets is an idempotent operation.

Next, we define the attribute, *self*, that for every *set*-object returns the set of all members of this set:

$$\begin{array}{l} nil[self \twoheadrightarrow \{\}] \\ add(X, L)[self \twoheadrightarrow X] \leftarrow L : set \\ add(X, L)[self \twoheadrightarrow Y] \leftarrow L : set[self \twoheadrightarrow Y] \end{array}$$

Finally, the *powerset* method is defined as follows:

$$\begin{array}{lll} S\,[\,powerset \twoheadrightarrow nil\,] & \leftarrow & S : set \\ S\,[\,powerset \twoheadrightarrow add(Z, L)\,] & \leftarrow & S : set[self \twoheadrightarrow Z;\ powerset \twoheadrightarrow L] \\ & & \wedge\ L : set\ \wedge \neg L[self \twoheadrightarrow Z] \end{array}$$

The first clause says that the empty set is a member of any power-set. The second rule says that a set obtained from a subset, L, of S by adding an element $Z \in S - L$, is a subset of S and thus is also a member of the power-set of S.

11.4.2 Querying Database Schema

The higher-order syntax of F-logic makes it possible to query and manipulate certain meta-information about the database, such as its schema.

Schema querying is the subject of this section. This issue was also discussed in [55] and recently in [30]. However, the treatment in [55] is not as general and integrated as in F-logic, while [30] is a relational rather than an object-oriented language.

However, this gain in efficiency has a price. For instance, in LDL, comparing sets for equality is easy because LDL treats each set as an entity and, in particular, the equality operator is applicable to sets. In contrast, F-logic does not have a built-in equality operator for sets. Nevertheless, comparing sets is possible by defining set-equality and set-containment predicates via logical rules.

To see how, suppose we want to verify that the result returned by one set-valued attribute stands in a certain relation (*e.g.*, equals, subset, etc.) to the value of another attribute of some other object. Defining these relationships in F-logic is akin to comparing relations in classical logic programming:

$$\begin{aligned}
notsubset(Obj, Attr, Obj', Attr') &\leftarrow Obj[Attr \twoheadrightarrow X] \wedge \neg Obj'[Attr' \twoheadrightarrow X] \\
subset(Obj, Attr, Obj', Attr') &\leftarrow \neg notsubset(Obj, Attr, Obj', Attr') \\
unequal((Obj, Attr, Obj', Attr') &\leftarrow notsubset(Obj, Attr, Obj', Attr') \\
unequal((Obj, Attr, Obj', Attr') &\leftarrow notsubset(Obj', Attr', Obj, Attr) \\
equal(Obj, Attr, Obj', Attr') &\leftarrow \neg unequal((Obj, Attr, Obj', Attr')
\end{aligned}$$

It should be noted, however, that although expressing set-equality in F-logic is more cumbersome than in LDL, implementing the equality operator of LDL involves the use of negation as failure, too. Therefore, comparing sets in F-logic and LDL has the same complexity.

Data restructuring. The next example is an adaptation from [3]. The issue here is the representation of data functions of COL [2] and grouping of LDL [16].

Consider a relation, $graph(X, Y)$, whose tuples represent edges of a graph. The task is to re-structure this graph by representing it as a set of nodes, such that each node points to a set of its descendants. The corresponding F-program is very simple:

$$rebuiltGraph[nodes \twoheadrightarrow Node[descendants \twoheadrightarrow D]] \leftarrow graph(Node, D)$$

where $rebuiltGraph$ is a new object. This rule also shows one more way to do nesting. This time, though, we are nesting a relation, $graph$, rather than a class.

The reader familiar with HiLog [30] may note that this rule can be made more general if we extend the syntax of F-logic to include variables that range over predicate names:

$$rebuilt(Rel)[nodes \twoheadrightarrow Node[descendants \twoheadrightarrow D]] \leftarrow Rel(Node, D) \wedge Rel : relation$$

where Rel is a variable and $relation$ is an appropriately defined class of binary relations. This rule will restructure any binary relation that is passed as a parameter. For instance,

$$?- rebuilt(yourFavouriteRel)[nodes \twoheadrightarrow Node[descendants \twoheadrightarrow D]]$$

will return a rebuilt version of $yourFavouriteGraph$.

For another example, suppose that we have a method, *grade*, declared as follows:

$$course[grade@student \Rightarrow integer]$$

A method like this would be appropriate for the use by an administrator. However, students are likely to be denied access to this method in class *course* because this will expose *grades* of other students. On

11.4.1 Set Manipulation

The ability to manipulate sets with ease is an important litmus test for an object-oriented data language. This subsection illustrates set-manipulation by expressing a number of popular set-manipulation operators, such as nesting and unnesting, set-comparison, and powerset operator. Other examples, such as set intersection and union operators can be found in [52].

Nesting and unnesting. Among the many set-related operations, the ability to restructure objects by nesting and unnesting are among the most important ones. Specifying these operations in F-logic is quite easy. Consider a class of objects with the following structure:

$$c[attr_1 \Rightarrow r_1; attr_2 \Rightarrow r_2]$$

One way to nest this class, say, on $attr_2$, is by defining a new class, $nest(c, attr_2)$, with the signature

$$nest(c, attr_2)[attr_1 \Rightarrow r_1; attr_2 \Rightarrow\!\!\!> r_2]$$

This class is populated according to the following rule:

$$nested(Y) : nest(c, attr_2)[attr_1 \rightarrow Y; attr_2 \twoheadrightarrow Z] \;\leftarrow\; X : c\,[attr_1 \rightarrow Y; attr_2 \rightarrow Z]$$

It is easy to see from either the semantics or the proof theory that this rule has the following meaning: to nest c on $attr_2$, populate the class $nest(c, attr_2)$ with the objects of the form $nested(y)$, such that their attribute $attr_2$ groups all z's that occur with y in some object x in c.

Similarly, consider a class with the following signature:

$$c[attr_1 \Rightarrow r_1; attr_2 \Rightarrow\!\!\!> r_2]$$

To unnest this class on the attribute $attr_2$, we define a new class, $unnest(c, attr_2)$, with the following signature:

$$unnest(c, attr_2)[attr_1 \Rightarrow r_1; attr_2 \Rightarrow r_2]$$

Identities of objects in this class depend on both the source-objects (from class c) and on the values returned by $attr_2$:

$$unnested(X, Z) : unnest(c, attr_2)[attr_1 \rightarrow Y; attr_2 \rightarrow Z] \;\leftarrow\; X : c\,[attr_1 \rightarrow Y; attr_2 \twoheadrightarrow Z]$$

This rule says that to unnest c on $attr_2$, we must define new objects of the form $unnested(x, z)$, for each c-object, x, and for each value z of $attr_2$ on x. In the unnested objects, both attributes, $attr_1$ and $attr_2$, are scalar and are defined so as to flatten the structure of the original objects in class c.

Set comparison. Grouping is not only easy to express in F-logic, but it is also computationally more efficient than in some other languages, such as LDL [77], COL [2], or LPS [59]. The reason for this is that, in these languages, grouping is a second-order operation that requires stratification. We refer the reader to [52] for a more complete discussion.

11.3 Examples of Type Declarations

Typing is a popular concept in programming languages; in its primitive form it is used in traditional database systems under the disguise of schema declaration. In programming, typing can be very useful both as a debugging aid and as a means of maintaining data integrity. It allows the user to define correct usage of objects and then let the system detect ill-typed data and queries. The purpose of type declarations is to impose type-constraints on arguments of methods as well as on the results returned by the methods. For instance, in

$$empl[salary@year \Rightarrow integer]$$
$$person[birthdate \Rightarrow year]$$

the first molecule states that *salary* is a function that for any *empl*-object would return an object in the class *int*, if invoked with an argument of class *year*. The second clause says that *birthdate* is an attribute that returns a year for any *person*-object. Section 12 discusses how typing constraints are imposed in F-logic. In the present section we shall only give some examples of type definitions, leaving the semantic considerations till later.

In the previous subsection, we have seen an examples of a parametric family of classes, such as $list(T)$. A signature for this family can be given as follows:

$$list(T)[first \Rightarrow T;\ rest \Rightarrow list(T);\ length \Rightarrow int;\ append@list(T) \Rightarrow list(T)] \tag{11}$$

These signatures are parametric; they declare the attributes *first*, *rest*, *length*, and the method *append* as polymorphic functions that can take arguments of different type. As with any clause in a logic program, variables in (11) are universally quantified.

For instance, the signature for *append* says that if it is invoked on a $list(int)$-object with an argument $list(int)$ then the result must be an object of class $list(int)$. However, if the argument is, say, $list(real)$ then the output must be an object of class $list(real)$, since $int :: real$. This is because this invocations of *append* must conform to the signature $list(real)[append@list(real) \Rightarrow list(real)]$, which is an instance of (11). Note that the output of this invocation does not have to be in class $list(int)$, because the signature-instance $list(int)[append@list(int) \Rightarrow list(int)]$ does not cover the invocation in question (here *append* is invoked on a list of reals, not integers).

11.4 Examples of Object Bases

This section illustrates the expressive power of F-logic on a number of interesting and non-trivial examples, which include manipulation of set and database schema, analogical reasoning, list processing, and others. In many cases, we shall omit signatures because they are simple and do not illustrate new ideas (except for the list processing methods whose typing has already been discussed). Also, since inheritance of properties will be discussed separately, all data expressions used in the examples, below, are of non-inheritable variety.

In many applications, the IS-A hierarchy may depend on other data, which commonly happens in view definitions, such as derived classes. For instance,

$$Car : dieselCars(Year) \leftarrow Car : car[engineType \rightarrow \text{“}diesel\text{”}; makeYear \rightarrow Year]$$

defines a family of derived classes, parameterized by $Year$. Each class, *e.g.*, $dieselCars(1990)$, would be inhabited by objects that represent diesel cars made in 1990.

Furthermore, sometimes it is desirable to be able to create derived classes whose population is determined by the objects' structure rather than by their properties, as it was in the previous example. For instance, the rule

$$X : merchandise \leftarrow X[price \Rightarrow ()]$$

defines a derived class, *merchandise*, that consists of all objects to which the attribute *price* applies. Note that an object would fall into the class *merchandise* if it has the attribute *price*, even if the price for that merchandise has not been set yet.

Our last example concerns the set-theoretic and the lattice-theoretic operators on the class hierarchy. F-logic does not require classes to form a lattice, *i.e.*, a pair of classes, c and c', does not have to have the lowest superclass or the greatest subclass. It is not even required that c and c' will have an intersection (or a union) class, *i.e.*, a class whose extension (set of members) is in an intersection (resp., union) of the extensions of c and c'. However, we can *define* class constructors to accomplish these tasks. For instance, the following rules define $and(X, Y)$ and $or(X, Y)$ to be intersection and union classes of its arguments, X and Y:

$$\begin{array}{lcl} I : or(X, Y) & \leftarrow & I : X \\ I : or(X, Y) & \leftarrow & I : Y \\ I : and(X, Y) & \leftarrow & I : X \wedge I : Y \end{array}$$

Note that, in the canonical model, $and(X, Y)$ is not a subclass of X and Y, and neither is $or(X, Y)$ their superclass. The above rules relate only *extensions* of the classes involved, not the classes themselves. If we also wanted to relate the classes, we would write:

$$\begin{array}{lcl} lsup(X, Y) :: C & \leftarrow & X :: C \wedge Y :: C \\ X :: lsup(X, Y) & & \\ Y :: lsup(X, Y) & & \\ C :: gsub(X, Y) & \leftarrow & C :: X \wedge C :: Y \\ gsub(X, Y) :: X & & \\ gsub(X, Y) :: Y & & \end{array}$$

Here *lsup* is a constructor that defines the lowest superclass of X and Y, and *gsub* defines greatest subclasses. Note that, for instance, $gsub(X, Y)$ is *not* an intersection of the extensions of the classes X and Y: the extension of $gsub(X, Y)$ is a subset of the extensions of X and Y, but the latter may have common members that are not members of $gsub(X, Y)$. In other words, *gsub* and *lsup* construct lower and upper bounds in the class hierarchy, but not in the hierarchy of class extensions. However, by combining the above rules for *and* and *gsub* (and for *or* and *lsup*) it is easy to define constructors of lower and upper bounds in both hierarchies.

Informally, an object-base definition specifies what each method is supposed to do. Object definitions may be explicit, *i.e.*, given as facts, or implicit, *i.e.*, specified via deductive rules. Class-hierarchy declarations, as their name suggests, organize objects and classes into IS-A hierarchies. Signature declarations specify the types of the arguments for each method and the type of the output they produce.

11.2 Examples of IS-A Hierarchies

Given an F-program and its canonic H-model, the IS-A hierarchy defined by the program is the set of all is-a atoms satisfied by the canonic model.

One simple example of a class-hierarchy declaration is given in Figures 2 and 3 of Section 3. In this hierarchy, *john* is a student and an employee at the same time; *phil* is an employee but not a student. The latter is *not* stated explicitly and *cannot* be derived using the normal logical implication, "$\models$", of Section 5. However, $\neg phil : student$ holds in the minimal model of the program of Section 3. Since the semantics is determined by this model, $\neg phil : student$ is considered as a valid statement about the class hierarchy.

The idea of class hierarchies is an important ingredient in the phenomenon known as *inclusion polymorphism*. For instance, stating that students and employees are persons implies that all properties (*i.e.*, methods) applicable to persons must automaticly be applicable to employees and students. In Figure 4 of Section 3, such properties are *name* (a scalar 0-ary method that returns objects of class *string*), *friends* (a set-valued attribute that returns objects of class *person*), and so on. In F-logic, inclusion polymorphism is built into the semantics. It is manifested by a property discussed in Section 7.3, called *structural inheritance*.

In F-logic, classes are objects and so they are represented by ground id-terms. This opens a way to represent parametric families of classes using non-ground id-terms, which gives the flexibility needed to support *parametric polymorphism*. For example, the following pair of clauses defines a parametric polymorphic type $list(T)$:

$$\begin{array}{l} nil : list(T) \\ cons(X, Y) : list(T) \leftarrow X : T \wedge Y : list(T) \\ list(T) :: list(S) \leftarrow T :: S \end{array} \qquad (10)$$

Here $list(T)$ denotes a parametric family of classes of the form

$$\{\, list(t) \mid \text{ where } t \text{ is a ground id-term } \}$$

For instance, if *int* denotes the class of integers then $list(int)$ is the class of lists of integers containing the elements nil, $cons(0, nil)$, $cons(0, cons(1, nil))$, and so on. The last clause above states that if T is a subclass of S then $list(T)$ is a subclass of $list(S)$. Note that this does not follow from the rest of the definition in (10) and has to be stated explicitly (if this property of lists is wanted, of course).

The above family of list-classes will be used later to define parameterized types for list-manipulation methods.

Then $\neg(bob \doteq dad(john))$ must hold in all canonic models of the program. However, since $bob \doteq dad(john)$ is a logical consequence of (9), this program has no equality-restricted canonic model.

Queries

A *query* is a statement of the form $?- Q$, where Q is a molecule.[16] The set of *answers* to $?- Q$ with respect to an F-program, $\mathbf{P}$, is the smallest set of molecules that

- contains all instances q of Q that are found in the canonic model of $\mathbf{P}$; and
- is closed under "$\models$".

The first condition is obvious and does not need further comments. The second condition is needed for rather technical reasons: Suppose the database contains $john[children \twoheadrightarrow \{bob, sally\}]$ and the query is $?- john[children \twoheadrightarrow X]$. Then there are two instances of the query implied by this database: $john[children \twoheadrightarrow bob]$ and $john[children \twoheadrightarrow sally]$. However, without $\models$-closedness, $john[children \twoheadrightarrow \{bob, sally\}]$ would not be considered an answer to the query, even though it is a logical consequence of the first two answers. On the other hand, $john[children \twoheadrightarrow \{bob, sally\}]$ would be viewed as an answer to a logically equivalent query, $?- john[children \twoheadrightarrow \{X, Y\}]$.

Closure with respect to "$\models$" eliminates this anomaly. It also makes it easier to talk about query equivalence and containment without getting bogged down in minor syntactic differences that may occur in essentially similar queries.

The Structure of F-logic Programs

F-programs specify what each method is supposed to do, define method signatures, and organize objects along class hierarchies. Thus, every program can be split into three disjoint components, according to the type of information they specify:

- The *IS-A hierarchy declaration.* This part of the F-program consists of the rules whose head-literal is an is-a assertion.
- The *signature declaration.* This part contains rules whose head literal is an object-molecule that is built out of signature expressions only.
- The *object-base definition.* This part consists of rules whose head literals do not contain signatures or is-a expressions, *i.e.*, rules whose heads are either predicates or object-molecules built exclusively out of data expressions.

Note that the above classification considers rule-heads only. It is legal, therefore, for rules in one component to have body-literals defined in other components. In fact, as we shall see, certain applications may need such flexibility.

[16] This does not limit generality, as every query can be reduced to this form by adding appropriate rules.

Canonic Models and Equality

Equality has always been a thorny issue in logic programming because it does not easily succumb to efficient treatment. As a result, most logic programming systems—and all commercial ones—have so called "freeness axioms" built into them.[15] In practice, this boils down to the restriction that prohibits equality predicates from appearing in the heads of program clauses.

Clearly, programming in F-logic cannot avoid such problems. Furthermore, equality has a much more prominent role in object-oriented programming than in classical logic programming. In F-logic, for instance, equations can be generated implicitly, without putting equality in the rule-heads. For example, consider the following simple F-program:

$$john[father \rightarrow bob] \qquad john[father \rightarrow dad(john)] \tag{9}$$

Because *father* is a scalar attribute, this program entails $bob \doteq dad(john)$. Likewise, $car :: automobile$ and $automobile :: car$ together entail $car \doteq automobile$ because the class hierarchy must be acyclic.

Apart from the usual, computational problems associated with the equality, implicit generation of equations may be problematic from the practical point of view. Indeed, multiply-defined scalar methods, such as *father* above, or cycles in the class-hierarchy may be unintentional, a result of programmer's mistake. Therefore, it may be desirable to regard certain programs, such as (9) above, as inconsistent, unless $dad(john) \doteq bob$ is also defined explicitly. Of course, since the above program obviously has many models, "inconsistency" here should be taken to mean the absence of *canonic* models. Furthermore, the notion of canonic models needs special adjustment for equality, because many programs that generate implicit equality may well have canonic models in the usual sense. For instance, (9) certainly has a minimal H-model, which consists of the two given atoms, the equation $bob \doteq dad(john)$, and tautologies.

The canonic models that take special care of the equality will henceforth be called *equality-restricted canonical models*, which is defined next.

The first step is to split each program, $\mathbf{P}$, into two (not necessarily disjoint) components, $\mathbf{P}^{\doteq}$ and $\mathbf{P}^{rest}$. The *equality definition*, $\mathbf{P}^{\doteq}$, is the part that determines the equality theory of $\mathbf{P}$. The second component, $\mathbf{P}^{rest}$, describes the rest.

For simplicity, we give a definition that relies on the assumption that the equality definition, $\mathbf{P}^{\doteq}$, has only one canonic H-model. However, it does not assume any specific theory of canonic models. Let $\mathbf{L}^{\doteq}$ be the (usual) canonical H-model of $\mathbf{P}^{\doteq}$ and let EQ be the set of all non-trivial equations in $\mathbf{L}^{\doteq}$. (An equation, $s \doteq t$, is non-trivial if and only if s and t are not identical.)

Definition 11.1 (Equality-restricted Canonical Models) Let $\mathbf{M}$ be a canonical H-model of $\mathbf{P}$ (we do not assume that $\mathbf{M}$ is unique). We say that $\mathbf{M}$ is an *equality-restricted canonical model* of $\mathbf{P}$ if the set of nontrivial equations in $\mathbf{M}$ coincides with EQ. □

Coming back to our example, suppose that $bob \doteq dad(john)$ is not in the equality-defining part of (9).

[15] In logic programming, freeness axioms are also known under the name of Clark's Equality Theory.

11.1 Logic Programs and their Semantics

Perhaps the most popular classes of logic programs is the class of *Horn programs*. A Horn F-program consists of *Horn rules*, which are clauses of the form

$$head \longleftarrow body \tag{8}$$

where *head* is an F-molecule and *body* is a conjunction of F-molecules. Since (8) is a clause, this implies that all its variables are implicitly universally quantified.

Just as in classical theory of logic programs, it is easy to show that Horn programs have the *model-intersection property.* Thus, the intersection of all H-models of a Horn program, $\mathbf{P}$, is also an H-model of $\mathbf{P}$. This model is also the *least* H-model of $\mathbf{P}$ with respect to set-inclusion (recall that H-structures are sets of molecules).

By analogy with classical theory of Horn logic programs (see, *e.g.*, [64]), we can define an F-logic counterpart of the well-known $T_{\mathbf{P}}$ operator that, given a Horn F-program $\mathbf{P}$, maps H-structures of $\mathbf{P}$ to other H-structures of $\mathbf{P}$. Given an H-structure, $\mathbf{I}$, $T_{\mathbf{P}}(\mathbf{I})$ is defined as the smallest H-structure that contains the following set:[13]

$$\{head \mid head \longleftarrow l_1 \wedge \cdots \wedge l_n \text{ is a ground instance of a rule in } \mathbf{P} \text{ and } l_1, ..., l_n \in \mathbf{I}\}$$

Following a standard recipe, it is easy to prove that the least fixpoint of $T_{\mathbf{P}}$ coincides with the least H-model of $\mathbf{P}$ and that a ground F-molecule, φ, is in the least H-model of $\mathbf{P}$ if and only if $\mathbf{P} \models \varphi$.

Although Horn programs can be used for a large number of applications, their expressive power is limited. For more expressiveness, it is necessary to relax the restrictions on the form of the rules and (referring to (8) above) allow negated F-molecules in *body*. Such programs will be called *generalized* (or *normal*) F-programs.

For generalized programs, the elegant connection between fixpoints, minimal models, and logical entailment holds no more. In fact, such programs may have several minimal models and the "right" choice is not always obvious. However, it is generally accepted that the semantics of a generalized logic program is given by the set of its *canonic* models, which is a subset of the set of all models of the program. Alas, in many cases there is no agreement as to which models deserve to be called canonic. Nevertheless, for a vast class of programs, called *locally stratified* programs, such an agreement had been reached, and it was shown in [81] that every such program has a unique canonic H-model.[14] For a locally stratified program, its unique canonic model goes under the name *perfect model.*

In Appendix A, we propose a perfect-model semantics for locally stratified F-programs, which is an adaptation from [81]. For our current needs, however, we shall assume that some canonic H-model exists for each F-program under consideration; details of these models will be immaterial for the discussions that follow. Moreover, since most of our examples are based on Horn programs, the canonic models of these programs coincide with their unique minimal models.

[13] By itself, this set may not be an H-structure because of the closure properties that an H-structure must satisfy (see Section 8).

[14] We consider only *definite* rules, *i.e.*, rules that do not have disjunctions in the head.

denote the derivation sequence that refutes $\mathbf{T}_2 = S' \cup \{L\}$ (which exists by the inductive assumption). Since $\mathbf{S}' \subset \mathbf{S}$ and $\mathbf{S} \vdash L$, it follows that if we apply $dedseq_1$ to $\mathbf{S}$ and then follow this up with steps from $dedseq_2$, we shall refute $\mathbf{S}$. □

Proposition 10.6 *There exists a unification algorithm that, given a pair of molecules T_1 and T_2, yields a complete set of mgu's of T_1 into T_2.*

Proof: The algorithm is given in Appendix B, and its correctness is proved in Lemma B.1. □

Lemma 10.7 (Lifting Lemma) *Suppose C_1, C_2 are clauses and C_1', C_2' are their instances, respectively. If D' is derived from C_1' and C_2' (or from C_1' alone) using one of the derivation rules, then there exists a clause, D, such that*

- *D is derivable from the* factors *of C_1 and C_2 (resp., from C_1 alone) via a single derivation step;*
- *This derivation step uses the same inference rule as the one that derived D'; and*
- *D' is an instance of D.*

Proof: Consider each derivation rule separately. The proof in each case is similar to that in predicate calculus since the notion of substitution is the same in both logics. Simple (but tedious) details are left as an exercise. □

Theorem 10.8 (Completeness of F-logic Inference System) *If a set* **S** *of clauses is unsatisfiable, then there is a refutation of* **S**.

Proof: The proof is standard. Consider $\mathbf{S}^*$, the set of all ground instances of **S**. By the ground case (Theorem 10.5), there is a refutation of $\mathbf{S}^*$. With the help of Lifting Lemma, this refutation can be then lifted to a refutation of **S**. □

11 Data Modeling in F-logic

In this section, we define the notions of a logic program, a database, and a query, and then illustrate the use of F-logic on a number of simple, yet non-trivial examples. We shall use the terms "deductive database" (or simply a database) and "logic program" interchangeably. As a first cut, we could say that a logic program in F-logic (abbr., an *F-program*) is an arbitrary set of F-formulae. However, as in classical logic programming, this definition is way too general and both pragmatic and semantic considerations call for various restrictions on the form of the allowed formulas.

(i) *P is an is-a assertion or a predicate:*
If P is an is-a assertion, item (2) in the construction of $\mathbf{M}$ in Section 8 can be used to show that $\mathbf{M} \models P$ if and only if $P \in \mathbf{D}(\mathbf{S})$. If P is a predicate, item (9) can be used to show the same.

(ii) *P is an object molecule composed of atoms* $\tau_1, ..., \tau_n$*:*
Then $\mathbf{M} \models P$ if and only if $\mathbf{M} \models \tau_i$, $i = 1, ..., n$. By items (5), (6), (7), or (8) of Section 8 (depending on whether the method expression in τ_i is scalar or set-valued and whether it is a data or a signature expression), it follows that $\mathbf{M} \models \tau_i$ if and only if $\tau_i \in \mathbf{D}(\mathbf{S})$. Therefore, by the definition of $\mathbf{D}(\mathbf{S})$, there are molecules $Q_1, ..., Q_n$ deducible from $\mathbf{S}$, such that τ_i is a submolecule of Q_i, for $i = 1, ..., n$. Let Q be the canonical merge of $Q_1, ..., Q_n$. Then P is a submolecule of Q (since every constituent atom of P is also a constituent atom of Q) and Q is deducible from $\mathbf{S}$ (since $Q_1, ..., Q_n$ are deducible from $\mathbf{S}$ and Q is obtained from $Q_1, ..., Q_n$ by the merging rule). Hence, P is in $\mathbf{D}(\mathbf{S})$, which proves (7).

By the definition of $\mathbf{D}(\mathbf{S})$, if $P \in \mathbf{S}$ is a positive literal then $P \in \mathbf{D}(\mathbf{S})$. Hence, by (7), $\mathbf{M} \models P$. For every negative literal $\neg P$ in $\mathbf{S}$, P is *not* a submolecule of any molecule deducible from $\mathbf{S}$, by the assumption made at the beginning of the proof. So, P is not in $\mathbf{D}(\mathbf{S})$. Again, by (7), $\mathbf{M} \not\models P$ and therefore $\mathbf{M} \models \neg P$. Thus, $\mathbf{M}$ satisfies every literal of $\mathbf{S}$, that is, it is a model for $\mathbf{S}$. □

Theorem 10.5 (Completeness of ground deduction) *If a set of ground clauses,* $\mathbf{S}$*, is unsatisfiable then there exists a refutation of* $\mathbf{S}$*.*

Proof: By Herbrand's Theorem, we may assume that $\mathbf{S}$ is finite. Suppose $\mathbf{S}$ is unsatisfiable. We will show that there is a refutation of $\mathbf{S}$ using a technique due to Anderson and Bledsoe [8]. The proof is carried out by induction on the parameter $excess(\mathbf{S})$, the number of "excess literals" in $\mathbf{S}$:

$$excess(\mathbf{S}) \stackrel{\text{def}}{=} (\text{ the number of occurrences of literals in } \mathbf{S}) - (\text{the number of clauses in } \mathbf{S}).$$

Basis: $excess(\mathbf{S}) = 0$. In this case, the number of clauses in $\mathbf{S}$ equals the number of occurrences of literals in $\mathbf{S}$. Hence either $\Box \in \mathbf{S}$ and we are done, or every clause in $\mathbf{S}$ is a literal. In the latter case, by Lemma 10.4, $\mathbf{S} \vdash \neg P$ and $\mathbf{S} \vdash Q$ for some molecules P, Q such that $P \sqsubseteq Q$. Applying the resolution rule to $\neg P$ and Q, we obtain the empty clause.

Induction Step: $excess(\mathbf{S}) = n > 0$. In this case, there must be a clause C in $\mathbf{S}$ that contains more than one literal. Let us separate this clause from the other clauses and write $\mathbf{S} = \{C\} \cup \mathbf{S}'$, where $C = L \vee C'$ ($C' \neq \Box$ since we have assumed that C contains more than one literal). By the distributivity law, $\{L \vee C'\} \cup \mathbf{S}'$ is unsatisfiable if and only if so are $\mathbf{T}_1 = \{C'\} \cup \mathbf{S}'$ and $\mathbf{T}_2 = \{L\} \cup \mathbf{S}'$. Since $excess(\mathbf{T}_1) < n$ and $excess(\mathbf{T}_2) < n$, the induction hypothesis ensures that there are refutations of $\mathbf{T}_1$ and $\mathbf{T}_2$ separately. Therefore, $\mathbf{T}_1 \vdash \Box$, where $\Box$ is the empty clause. Let $dedseq_1$ denote the deduction sequence that derives $\Box$ from $\mathbf{T}_1$. Applying the deductive steps in $dedseq_1$ to $\mathbf{S}$, we can derive either L or $\Box$. If $\Box$ is so produced, then we are done. Otherwise, if L is produced, it means that $\mathbf{S} \vdash L$. Let $dedseq_2$

10.8 A Sample Proof

Consider the following set of clauses:

i.	$a :: b$	iv.	$r[attr \rightarrow a]$
ii.	$p(a)$	v.	$r[attr \rightarrow f(S)] \vee \neg p(X) \vee \neg O[M @ X \Rightarrow S]$
iii.	$c[m @ b \Rightarrow (v, w)]$	vi.	$\neg p(f(Z))$

We can refute the above set using the following sequence of derivation steps, where θ denotes the unifier used in the corresponding step:

vii.	$r[attr \rightarrow f(S)] \vee \neg O[M @ a \Rightarrow S]$	by resolving (ii) and (v); $\theta = \{X \backslash a\}$
viii.	$c[m @ a \Rightarrow (v, w)]$	by input-type restriction from (i) and (iii)
ix.	$r[attr \rightarrow f(v)]$	by resolving (viii) with (vii); $\theta = \{O \backslash c, S \backslash v, M \backslash m\}$
x.	$a \doteq f(v)$	by the rule of scalarity, using (iv) and (ix)
xi.	$p(f(v))$	by paramodulation, using (ii) and (x)
xii.	$\Box$	by resolving (vi) with (xi); $\theta = \{Z \backslash v\}$

All steps in this derivation are self-explanatory. We would like to point out, though, that θ in Step (ix) is an asymmetric unifier that unifies the atom $O[M @ a \Rightarrow S]$ *into* the molecule (iii).

10.9 Completeness of the Proof Theory

We follow the standard strategy for proving completeness, adapted from classical logic. First, Herbrand's Theorem is used to establish completeness for the ground case. Then, an analogue of Lifting Lemma shows that ground refutations can be "lifted" to the nonground case.

Lemma 10.4 *Let* $\mathbf{S}$ *be a set of ground F-literals. If* $\mathbf{S}$ *is unsatisfiable then there are molecules* P *and* Q *such that* $P \sqsubseteq Q$, $\neg P \in \mathbf{S}$ *and* $\mathbf{S} \vdash Q$. *(When* P, Q *are P-molecules or is-a assertions,* $P \sqsubseteq Q$ *should be taken to mean that* P *is identical to* Q*.)*

Proof: Suppose to the contrary, that there are no molecules P and Q, such that $P \sqsubseteq Q$, $\neg P \in \mathbf{S}$, and $\mathbf{S} \vdash Q$. We will show that then $\mathbf{S}$ must be satisfiable. Consider a set of molecules,

$$\mathbf{D}(\mathbf{S}) \stackrel{\text{def}}{=} \{P \mid \ P \text{ is a submolecule of some molecule } Q \text{ such that } \mathbf{S} \vdash Q \ \}.$$

Section 8 shows that every H-structure, $\mathbf{H}$, has a corresponding F-structure $\mathbf{I}_H$ such that $\mathbf{H} \models \mathbf{S}$ if and only if $\mathbf{I}_H \models \mathbf{S}$. Applying the same construction to $\mathbf{D}(\mathbf{S})$ we obtain an F-structure, $\mathbf{M}$. Since $\mathbf{D}(\mathbf{S})$ is closed under deduction, it is easy to verify that $\mathbf{M}$ is, indeed, an F-structure. We claim that for every molecule P:

$$\mathbf{M} \models P \quad \text{if and only if} \quad P \in \mathbf{D}(\mathbf{S}) \tag{7}$$

The "if"-direction follows from soundness of the derivation rules. For the "only if"-direction, consider the following two cases:

Scalarity:
$$\frac{P[Mthd@Q_1,...,Q_k \rightarrow R] \vee C_1, \quad P'[Mthd'@Q'_1,...,Q'_k \rightarrow R'] \vee C_2 \quad \theta = mgu(\langle P, Mthd, Q_1,...,Q_k\rangle, \langle P', Mthd', Q'_1,...,Q'_k\rangle)}{\theta(R \doteq R' \vee C_1 \vee C_2)}$$

Similarly for inheritable scalar expressions, where $\rightarrow$ is replaced with $\bullet\!\!\rightarrow$

Merging:
$$\frac{P[...] \vee C, \quad P'[...] \vee C', \quad \theta = mgu(P, P'), \quad L'' = merge(\theta(P[...]), \theta(P'[...]))}{L'' \vee \theta(C \vee C')}$$

Elimination:
$$\frac{\neg P[\,] \vee C}{C}$$

Figure 8: Summary of the Miscellaneous Inference Rules

10.6 Remarks

With such multitude of inference rules, a natural concern is whether there might be an efficient evaluation procedure for F-logic queries. The answer to this question is positive: F-logic queries can be evaluated, say, bottom-up and the optimization strategies developed for deductive databases (*e.g.*, Magic Sets and Templates [18, 83]) are applicable here as well.

Another important point is that one does not need to use some of the inference rules at run time. For instance, in proof-theoretic terms, the purpose of static type checking is to obviate the need in using the typing rules at run time. Likewise, a compile-time acyclicity-checking algorithm could be used to get rid of the IS-A acyclicity rule at run time. A practical system is also likely to limit the use of the rule of scalarity. For instance, this rule may be used to generate run-time warnings regarding possible inconsistencies detected in scalar methods, but not to do inference.

10.7 Soundness of the Proof Theory

Given a set **S** of clauses, a *deduction* of a clause C from **S** is a finite sequence of clauses $D_1, ..., D_n$ such that $D_n = C$ and, for $1 \leq k \leq n$, D_k is either

- a member of **S**, or
- is derived from some D_i (and, possibly, some additional clause D_j), where $i, j < k$, using one of the core, is-a, type, or miscellaneous inference rules.

A deduction ending with the empty clause, $\Box$, is called a *refutation* of **S**. If C is deducible from **S**, we shall write $\mathbf{S} \vdash C$.

Theorem 10.3 (Soundness of F-logic Deduction) *If* $\mathbf{S} \vdash C$ *then* $\mathbf{S} \models C$.

Proof: Directly follows from the closure properties given in Section 7 and from the form of the inference rules. $\Box$

A similar rule exists for inheritable scalar expressions. The only difference is that $\rightarrow$ is replaced with $\bullet\!\!\rightarrow$ in W and W'.

Another miscellaneous rule is called *merging*; it seeks to combine information contained in different object molecules. Let L_1 and L_2 be a pair of such molecules with the same object id. An object-molecule L is called a *merge* of L_1 and L_2, if the set of constituent atoms of L is precisely the union of the sets of constituent atoms of L_1 and L_2. A pair of molecules can be merged in several different ways when they have common set-valued methods. For example, the terms

$$T[ScalM \rightarrow d;\ SetM \twoheadrightarrow e;\ SetM @ X \twoheadrightarrow b] \tag{3}$$

$$T[ScalM \rightarrow g;\ SetM @ Y \twoheadrightarrow h;\ SetM @ X \twoheadrightarrow c] \tag{4}$$

have more than one merge:

$$T[ScalM \rightarrow d;\ ScalM \rightarrow g;\ SetM \twoheadrightarrow e;\ SetM @ Y \twoheadrightarrow h;\ SetM @ X \twoheadrightarrow b;\ SetM @ X \twoheadrightarrow c] \tag{5}$$

and

$$T[ScalM \rightarrow d;\ ScalM \rightarrow g;\ SetM \twoheadrightarrow e;\ SetM @ Y \twoheadrightarrow h;\ SetM @ X \twoheadrightarrow \{b, c\}] \tag{6}$$

However, we distinguish certain kind of merges that do have the uniqueness property. We call them *canonical* merges.

An *invocation* of a method consists of the method's name, its arguments, and the arrow specifying the type of invocation (scalar or set-valued). For instance, in the above example, $ScalM \rightarrow$, $SetM \twoheadrightarrow$, $SetM @ X \twoheadrightarrow$, and $SetM @ Y \twoheadrightarrow$ are all distinct invocations. A *canonical merge* of L_1 and L_2, denoted $merge(L_1, L_2)$, is a merge that does not contain repeated identical invocations of set-valued methods. In the above, (6) is a canonical merge of (3) and (4). Clearly, $merge(L_1, L_2)$ is unique up to a permutation of atoms and id-terms in the ranges of set-valued methods.

Merging: Consider a pair of standardized apart clauses, $W = L \vee C$ and $W' = L' \vee C'$, where both L and L' are object molecules. Let θ be an mgu unifying the oid parts of L and L'. Let L'' denote the canonical merge of $\theta(L)$ and $\theta(L')$. The *merging* rule, then, sanctions the following derivation:

$$\textbf{from}\ \ W\ \ \text{and}\ \ W'\ \ \textbf{derive}\ \ L'' \vee \theta(C \vee C')$$

Finally, since for every id-term P, the molecule $P[\,]$ is a tautology, we need the following *elimination* rule:

Elimination: If C is a clause and P an id-term then:

$$\textbf{from}\ \ \neg P[\,] \vee C\ \ \textbf{derive}\ \ C$$

Notice that if C is an empty clause then the elimination rule would derive an empty clause as well.

Type inheritance:
$$\frac{P[Mthd@Q_1,...,Q_k \Rightarrow T] \vee C,\ (S' :: P') \vee C',\ \theta = mgu(P,P')}{\theta(S'[Mthd@Q_1,...,Q_k \Rightarrow T] \vee C \vee C')}$$

Similarly for set-valued methods

Input restriction:
$$\frac{P[Mthd@Q_1,...,Q_i,...,Q_k \Rightarrow T] \vee C,\ \ (Q_i'' :: Q_i') \vee C',\ \ \theta = mgu(Q_i,Q_i')}{\theta(P[Mthd@Q_1,...,Q_i'',...,Q_k \Rightarrow T] \vee C \vee C')}$$

Similarly for set-valued methods

Output relaxation:
$$\frac{P[Mthd@Q_1,...,Q_k \Rightarrow R] \vee C,\ \ (R' :: R'') \vee C',\ \ \theta = mgu(R,R')}{\theta(P[Mthd@Q_1,...,Q_k \Rightarrow R''] \vee C \vee C')}$$

Similarly for set-valued methods

Figure 7: Summary of the Type-Inference Rules

Input-type restriction: Let $W = P[Mthd@Q_1,...,Q_i,...,Q_k \Rightarrow T] \vee C$ and $W' = (Q_i'' :: Q_i') \vee C'$ be standardized apart. Suppose also that Q_i and Q_i' have an mgu θ. The *input restriction* rule states:

$$\textbf{from}\ \ W\ \ \text{and}\ \ W'\ \ \textbf{derive}\ \ \theta(P[Mthd@Q_1,...,Q_i'',...,Q_k \Rightarrow T] \vee C \vee C')$$

Here Q_i'' replaces Q_i. A similar rule exists for set-valued methods.

Output-type relaxation: Consider clauses $W = P[Mthd@Q_1,...,Q_k \Rightarrow R] \vee C$ and $W' = (R' :: R'') \vee C'$ with no common variables, and suppose R and R' have an mgu θ. The *output relaxation* rule, then, states:

$$\textbf{from}\ \ W\ \ \text{and}\ \ W'\ \ \textbf{derive}\ \ \theta(P[Mthd@Q_1,...,Q_k \Rightarrow R''] \vee C \vee C')$$

Similar rules apply to set-valued methods.

We also note that in the first two inference rules above, T is either an id-term or $(\,)$.

10.5 Miscellaneous Inference Rules

The property that scalar methods return at most one value is enforced by the following rule:

Scalarity: Consider a pair of clauses that share no common variables:

$$W = P[Mthd@Q_1,...,Q_k \rightarrow R] \vee C \quad \text{and} \quad W' = P'[Mthd'@Q_1',...,Q_k' \rightarrow R'] \vee C'.$$

Suppose there is an mgu θ that unifies the tuple of id-terms $\langle P, Mthd, Q_1, ..., Q_k\rangle$ with the tuple $\langle P', Mthd', Q_1', ..., Q_k'\rangle$. The rule of *scalarity* then says:

$$\textbf{from}\ \ W\ \ \text{and}\ \ W'\ \ \textbf{derive}\ \ \theta(\,(R \doteq R') \vee C \vee C')$$

IS-A Reflexivity:	X::X
IS-A acyclicity:	$\dfrac{(P :: Q) \vee C,\ \ (Q' :: P') \vee C',\ \ \theta = mgu(\langle P, Q\rangle, \langle P', Q'\rangle)}{\theta(\ (P \doteq Q) \vee C \vee C')}$
IS-A transitivity:	$\dfrac{(P :: Q) \vee C,\ \ (Q' :: R') \vee C',\ \ \theta = mgu(Q, Q')}{\theta((P :: R') \vee C \vee C')}$
Subclass inclusion:	$\dfrac{(P : Q) \vee C,\ \ (Q' :: R') \vee C',\ \ \theta = mgu(Q, Q')}{\theta((P : R') \vee C \vee C')}$

Figure 6: Summary of the IS-A Inference Rules

follows:

$$\textbf{from } W \text{ and } W' \textbf{ derive } \theta((P \doteq Q) \vee C \vee C')$$

Note that IS-A reflexivity and IS-A acyclicity imply reflexivity of equality. Indeed, since $X :: X$ is an axiom, by IS-A acyclicity, one can derive $X \doteq X$ from $X :: X$ and $X :: X$.

IS-A transitivity: Let $W = (P :: Q) \vee C$ and $W' = (Q' :: R') \vee C'$ be standardized apart and let θ be an mgu of Q and Q'. The *transitivity* rule, then, is:

$$\textbf{from } W \text{ and } W' \textbf{ derive } \theta((P :: R') \vee C \vee C')$$

Subclass inclusion: Let $W = (P : Q) \vee C$ and $W' = (Q' :: R') \vee C'$ be standardized apart and let θ be an mgu of Q and Q'. Then the *subclass inclusion* rule says:

$$\textbf{from } W \text{ and } W' \textbf{ derive } \theta((P : R') \vee C \vee C')$$

10.4 Type Inference Rules

Signature expressions have the properties of type inheritance, input-type restriction, and output-type relaxation that are captured by the following inference rules:

Type inheritance: Let $W = P[Mthd @ Q_1, ..., Q_k \Rightarrow T] \vee C$ and $W' = (S' :: P') \vee C'$ be a pair of clauses with no common variables, and suppose P and P' have an mgu, θ. The *type inheritance* rule states the following:

$$\textbf{from } W \text{ and } W' \textbf{ derive } \theta(S'[Mthd @ Q_1, ..., Q_k \Rightarrow T] \vee C \vee C')$$

In other words, S' inherits the signature of P'. A similar rule exists for set-valued methods. If $W = P[Mthd @ Q_1, ..., Q_k \Rightarrow\!\!> T] \vee C$ and W' is as before, then:

$$\textbf{from } W \text{ and } W' \textbf{ derive } \theta(S'[Mthd @ Q_1, ..., Q_k \Rightarrow\!\!> T] \vee C \vee C')$$

Resolution:	$\dfrac{\neg L \vee C,\ \ L' \vee C',\ \ \theta = mgu_{\sqsubseteq}(L, L')}{\theta(C \vee C')}$
Factoring:	$\dfrac{L \vee L' \vee C,\ \ \theta = mgu_{\sqsubseteq}(L, L')}{\theta(L \vee C)} \qquad \dfrac{\neg L \vee \neg L' \vee C,\ \ \theta = mgu_{\sqsubseteq}(L, L')}{\theta(\neg L' \vee C)}$
Paramodulation:	$\dfrac{L[T] \vee C,\ \ (T' \doteq T'') \vee C',\ \ \theta = mgu(T, T')}{\theta(L[T \backslash T''] \vee C \vee C')}$

Figure 5: Summary of the Core Inference Rules

Let L be unifiable *into* L' with the mgu θ. The *factoring* rule is, then, as follows:

$$\textbf{from } W \textbf{ derive } \theta(L \vee C)$$

In case of negative literals, if $W = \neg L \vee \neg L' \vee C$ and L is unifiable *into* L' with the mgu θ, then the factoring rule is:

$$\textbf{from } W \textbf{ derive } \theta(\neg L' \vee C)$$

Clauses inferred by one of the two factoring rules are called *factors* of W. Note that in both inference rules L must be unifiable into L'. However, in the first case, it is the literal L that survives, while in the second rule it is L'.

To account for the equality relation, we need a paramodulation rule. We use the following standard convention: When there is a need to focus on a specific occurrence of an id-term, T, in an expression, E (which can be a literal or an id-term), we may write $E[T]$. If one single occurrence of T is replaced by S, the result will be denoted by $E[T \backslash S]$.

Paramodulation: Consider a pair of clauses, $W = L[T] \vee C$ and $W' = (T' \doteq T'') \vee C'$, with no common variables. If T and T' are id-terms unifiable with an mgu, θ, then the *paramodulation* says:

$$\textbf{from } W \text{ and } W' \textbf{ derive } \theta(L[T \backslash T''] \vee C \vee C')$$

10.3 IS-A Inference Rules

The following axiom and the rules capture the semantics of the subclass relationship and its interaction with class membership.

IS-A Reflexivity: The following is the *IS-A reflexivity* axiom:

$$(\forall X)\ X :: X.$$

IS-A acyclicity: Let $W = (P :: Q) \vee C$ and $W' = (Q' :: P') \vee C'$ be clauses with no variables in common. Suppose that θ is an mgu of tuples $\langle P, Q \rangle$ and $\langle P', Q' \rangle$ of id-terms. The *acyclicity* rule is as

In predicate calculus, the notion of a unifier works for an arbitrary number of terms to be unified. Extension of our definitions to accommodate an arbitrary number of id-terms, P-molecules, or is-a assertions is obvious. For object molecules, we say that a substitution σ is a unifier of L_1, ..., L_n into L if and only if $\sigma(L_i) \sqsubseteq L$, for $i = 1, ..., n$. Generalization of the notion of mgu is straightforward and is left as an exercise.

For notational convenience, we also define mgu's for tuples of id-terms. Tuples $\langle P_1, ..., P_n \rangle$ and $\langle Q_1, ..., Q_n \rangle$ are *unifiable* if and only if there is a substitution σ such that $\sigma(P_i) = \sigma(Q_i)$, $i = 1, ..., n$. This unifier is *most general* (written $mgu(\langle P_1, ..., P_n \rangle, \langle Q_1, ..., Q_n \rangle)$), if and only if for every other unifier, μ, of these tuples, $\mu = \gamma \circ \sigma$ for some substitution γ. It is easy to see that any mgu of $\langle P_1, ..., P_n \rangle$ and $\langle Q_1, ..., Q_n \rangle$ coincides with the mgu of $f(P_1, ..., P_n)$ and $f(Q_1, ..., Q_n)$, where f is an arbitrary n-ary function symbol, and, therefore, is unique.

In the sequel, $mgu_{\sqsubseteq}(L_1, L_2)$ will be used to denote *some* most general unifier of one molecule, L_1, into another, L_2. As shown earlier, for object molecules $mgu_{\sqsubseteq}(L_1, L_2)$ may exist while $mgu_{\sqsubseteq}(L_2, L_1)$ may not. On the other hand, unification of id-terms, is-a assertions, and P-molecules is a symmetric operation. Nevertheless, we can still talk about unification of one such expression *into* another—a convention that can often simplify the language. Also, for the sake of simpler notation, all our inference rules will be based on F-atoms, not F-molecules.

Finally, we remark that, as in predicate calculus, prior to any application of an inference rule, the clauses involved in the application must be *standardized apart*. This means that variables must be consistently renamed so that the resulting clauses will share none. However, clauses used by an inference rule may be instances of the same clause; they can even be identical, if no variables are involved.

10.2 Core Inference Rules

For simplicity, but without loss of generality, only binary resolution is considered. In the inherence rules, below, the symbols L and L' will be used to denote F-literals (positive or negative), C and C' will denote clauses, and P, Q, R, S, T, etc., will denote id-terms.

Resolution: Let $W = \neg L \vee C$ and $W' = L' \vee C'$ be a pair of clauses that are standardized apart. Let θ be an mgu of L into L'. The *resolution* rule is, then, as follows:

$$\textbf{from} \quad W \quad \text{and} \quad W' \quad \textbf{derive} \quad \theta(C \vee C')$$

Notice that when L and L' are object molecules, resolution is *asymmetric* since $\theta = mgu_{\sqsubseteq}(L, L')$ may be different from $mgu_{\sqsubseteq}(L', L)$, and the latter mgu may not even exist. As in the classical case, binary resolution must be complemented with the so-called factoring rule that seeks to reduce the number of disjuncts in a clause.

Factoring: The factoring rule has two forms, depending on the polarity of literals to be factored. For positive literals, consider a clause of the form $W = L \vee L' \vee C$, where L and L' are positive literals.

Unifiers

Unification of id-terms, is-a assertions, and P-molecules is no different than in classical logic. Let T_1 and T_2 be a pair of id-terms, is-a molecules, or P-molecules. A substitution σ is a *unifier* of T_1 *and* T_2, if and only if $\sigma(T_1) = \sigma(T_2)$. This unifier is *most general*, written $mgu(T_1, T_2)$, if for every unifier μ of T_1 and T_2, there exists a substitution γ, such that $\mu = \gamma \circ \sigma$.

For object molecules, rather than requiring that they must become identical after applying a unifier, we ask merely that one molecule will be mapped into a *submolecule* of the other.

Definition 10.1 (Asymmetric Unification of Object Molecules) Let $L_1 = S[\ldots]$ and $L_2 = S[\ldots]$ be a pair of object molecules with the same object id, S. We say that L_1 is a *submolecule* of L_2, denoted $L_1 \sqsubseteq L_2$, if and only if every constituent atom of L_1 (defined in Section 7) is also a constituent atom of L_2. A substitution σ is a *unifier* of L_1 *into* L_2 (note the asymmetry!) if and only if $\sigma(L_1) \sqsubseteq \sigma(L_2)$. □

For instance, $L = S[M \mathbin{@} X \rightarrow V]$ is unifiable into $L' = S[N \mathbin{@} Y \rightarrow W; Z \mathbin{@} Y \bullet\!\!\rightarrow T]$ with the unifier $\{M\backslash N, X\backslash Y, V\backslash W\}$, but not the other way around (because the atom $S[Z \mathbin{@} Y \bullet\!\!\rightarrow T]$ cannot be turned into a constituent atom of L). However, a slightly different molecule, $S[N \mathbin{@} Y \rightarrow W; Z \mathbin{@} Y \rightarrow T]$, *is* unifiable into L with the unifier $\{N\backslash M, Y\backslash X, W\backslash V, Z\backslash M, T\backslash V\}$.

Complete Sets of Most General Unifiers

Defining *most general* unifiers for object molecules requires more work. Consider the terms $L_1 = a[set \twoheadrightarrow \{X\}]$ and $L_2 = a[set \twoheadrightarrow \{b, c\}]$. Intuitively, there are two unifiers of L_1 into L_2 that can be called "most general:" $X\backslash b$ and $X\backslash c$. Clearly, none of these unifiers is more general than the other and, therefore, the definition of mgu that works for P-molecules and for is-a assertions does not work here. A common approach in such situations is to consider *complete sets* of most general unifiers.

Definition 10.2 (Most General Unifiers) Let L_1 and L_2 be a pair of molecules and let α, β be a pair of unifiers of L_1 into L_2. We say that α is *more general* than β, denoted $\alpha \trianglelefteq \beta$, if and only if there is a substitution γ such that $\beta = \gamma \circ \alpha$. A unifier α of L_1 into L_2 is *most general* (abbr., *mgu*) if for every unifier β, $\beta \trianglelefteq \alpha$ implies $\alpha \trianglelefteq \beta$.

A set Σ of most general unifiers of L_1 into L_2 is *complete* if for every unifier θ of L_1 into L_2 there is $\alpha \in \Sigma$ such that $\alpha \trianglelefteq \theta$. □

Just as there is a unique-up-to-the-equivalence mgu in the classical case, it easily follows from the definitions that the complete set of unifiers of L_1 into L_2 is also unique up to the equivalence.[12] An algorithm that computes a complete set of mgu's appears in Appendix B.

[12] A set of unifiers, Ω_1, is *equivalent* to Ω_2 if for every $\sigma_1 \in \Omega_1$ there is $\sigma_2 \in \Omega_2$ such that $\sigma_2 \trianglelefteq \sigma_1$; and vice versa, for every $\sigma_2 \in \Omega_2$ there is $\sigma_1 \in \Omega_1$ such that $\sigma_1 \trianglelefteq \sigma_2$.

10 Proof Theory

This section describes a sound and complete proof theory for the logical entailment relation "$\models$" of Section 5. The theory consists of twelve inference rules and one axiom. The rules of resolution, factoring, and paramodulation form the core of the deductive system. However, unlike in predicate calculus, these three rules are not enough. For a deductive system to be complete, additional rules for capturing the properties of types and IS-A hierarchies are needed. The large number of inference rules in F-logic compared to predicate calculus stems from the rich semantics of object-oriented systems; this is likely to be the case with any logical system that attempts to adequately capture this paradigm. As will be seen shortly, many rules are quite similar to each other, except that one may deal with data expressions and the other with signatures or with P-molecules. It is possible to reduce the number of rules by about half through increasing the power of the resolution rule. This is analogous to classical logic, where factoring is often combined with resolution. However, we prefer to keep inference rules simple and increase their number instead, so that it would be easier to see the rationale behind each rule.

10.1 Substitutions and Unifiers

In F-logic, much of the theory of unifiers carries over from the classical case. However, this notion needs some adjustments due to the presence of sets and also because of the more flexible syntax of F-molecules (compared to classical logic).

Substitutions

Let $\mathcal{L}$ be a language with a set of variables $\mathcal{V}$. A *substitution* is a mapping $\sigma : \mathcal{V} \longmapsto \{\text{id-terms of } \mathcal{L}\}$ such that it is an identity everywhere outside some finite set $dom(\sigma) \subseteq \mathcal{V}$, the *domain* of σ.

As in classical logic, substitutions extend to mappings $\{\, \textit{id-terms} \,\} \longmapsto \{\, \textit{id-terms} \,\}$ as follows:

$$\sigma(f(t_1, ..., t_n)) = f(\sigma(t_1), ..., \sigma(t_n))$$

A substitution, σ, can be further extended to a mapping from molecules to molecules by distributing σ through the components of the molecules and applying it to each id-term. For instance, $\sigma(Q : P)$ is defined as $\sigma(Q) : \sigma(P)$, $\sigma(Q :: P)$ as $\sigma(Q) :: \sigma(P)$, and $\sigma(\, Q[Mthd @ R, S \rightarrow T]\,)$ is the same as $\sigma(Q)[\sigma(Mthd) @ \sigma(R), \sigma(S) \rightarrow \sigma(T)]$. Similarly, substitutions extend to F-formulae by distributing them through logical connectives and quantifiers.

A substitution is *ground* if $\sigma(X) \in U(\mathcal{F})$ for each $X \in dom(\sigma)$, that is, if $\sigma(X)$ has no variables. Given a substitution σ and a formula φ, $\sigma(\varphi)$ is called an *instance* of φ. It is a *ground* instance if it contains no variables. A formula is *ground* if it has no variables.

(i) For every ground F-molecule T, either $T \in \mathbf{T}$ or $\neg T \in \mathbf{T}$.

(ii) A ground clause, $L_1 \vee \cdots \vee L_n$, is in $\mathbf{T}$ if and only if $L_i \in \mathbf{T}$, for some $1 \leq i \leq n$.

Proof: Part (i): If $\mathbf{T}$ is finitely satisfiable, then either $\mathbf{T} \cup \{T\}$ or $\mathbf{T} \cup \{\neg T\}$ is finitely satisfiable. Therefore, $\mathbf{T}$ must contain either T or $\neg T$, since it is maximal. Part (ii) is proved similarly to (i). □

Lemma 9.4 *Let $\mathbf{T}$ be a maximal finitely satisfiable set of ground clauses. Let $\mathbf{H}$ be the set of all ground molecules in $\mathbf{T}$. Then $\mathbf{H}$ is an H-structure.*

Proof: It is easy to see that $\mathbf{H}$ is $\models$-closed, since $\mathbf{T}$ is also $\models$-closed (or else $\mathbf{T}$ is not maximal). □

Theorem 9.5 (cf. Herbrand's Theorem)
A set of clauses, $\mathbf{S}$, is unsatisfiable if and only if so is some finite subset of ground instances of the clauses in $\mathbf{S}$.

Proof: For the "if" part, assume that some finite subset of ground clauses of $\mathbf{S}$ is unsatisfiable. Then $\mathbf{S}$ is also unsatisfiable. The "only-if" part is proved by contradiction. Assume that some set of ground instances of the clauses in $\mathbf{S}$ is finitely satisfiable. We will show that then $\mathbf{S}$ is satisfiable.

Let $\mathbf{S}'$ be such finitely satisfiable set of ground instances. By Lemma 9.2, it can be extended to a maximal finitely satisfiable set $\mathbf{T}$. Let $\mathbf{H}$ be the set of all ground molecules in $\mathbf{T}$, which is an H-structure, by Lemma 9.4. We claim that $\mathbf{H} \models C$ if and only if $C \in \mathbf{T}$, for every ground clause, C. Consider the following cases:

(a) *C is a ground molecule.* By definition, $\mathbf{H} \models C$ if and only if $C \in \mathbf{T}$;

(b) *C is a negative literal $\neg P$.* Then $\mathbf{H} \models \neg P$ if and only if $P \notin \mathbf{H}$. Since $\mathbf{H}$ contains all the ground molecules in $\mathbf{T}$, $P \notin \mathbf{H}$ if and only if $P \notin \mathbf{T}$. Finally, by (i) of Lemma 9.3, $P \notin \mathbf{T}$ if and only if $\neg P \in \mathbf{T}$;

(c) *C is a disjunction of ground literals $L_1 \vee \cdots \vee L_n$.* Then
$\mathbf{H} \models L_1 \vee \cdots \vee L_n$
if and only if $\mathbf{H} \models L_i$ for some i, by definition;
if and only if $L_i \in \mathbf{T}$, by cases (a) and (b) above;
if and only if $L_1 \vee \cdots \vee L_n \in \mathbf{T}$, by (ii) of Lemma 9.3.

We have, thus, shown that $\mathbf{H}$ satisfies every clause of $\mathbf{T}$. Since $\mathbf{S}' \subseteq \mathbf{T}$, $\mathbf{H}$ is an H-model of $\mathbf{S}$. By Proposition 8.2, $\mathbf{S}$ is satisfiable. □

Herbrand's Theorem is a basis for the resolution-based proof theory in classical logic [27]. The next section presents a sound and complete resolution-based proof procedure, extending the result of [50]. This, in turn, provides a firm basis for a theory of object-oriented databases and programming.

Skolemization and Clausal Form

It is easy to verify that the usual De Morgan's laws hold for F-formulas. Therefore, every F-logic formula has a prenex normal form.

Once a formula is converted into an equivalent prenex normal form, it is Skolemized. Skolemization in F-logic is, again, similar to the classical case, since id-terms are identical to terms in predicate calculus and because quantification is defined similarly in both cases. For instance, a Skolem normal form for the molecule $(\forall X \exists Y) g(X,Y)[Y \rightarrow f(X,Y);\ a \twoheadrightarrow \{X,Y\}]$ would be $(\forall X) g(X,h(X))[h(X) \rightarrow f(X,h(X));\ a \twoheadrightarrow \{X,h(X)\}]$, where h is a new unary function symbol.

Theorem 9.1 (cf. Skolem's Theorem) *Let φ be an F-formula and φ' be its Skolemization. Then φ is unsatisfiable if and only if so is φ'.*

The proof of this theorem is almost identical to the standard proof for predicate calculus, and is omitted. From now on, we assume that all formulas are Skolemized. De Morgan's Laws further assure that every formula has a conjunctive and a disjunctive normal form. We can therefore transform every Skolemized formula into a logically equivalent set of clauses, where *clause* is a disjunction of literals.

Herbrand's Theorem

In classical logic, a set of clauses, $\mathbf{S}$, is unsatisfiable if and only if so is some finite set of ground instances of clauses in $\mathbf{S}$; this property is commonly referred to as *Herbrand's theorem*. In F-logic, Herbrand's theorem plays the same fundamental role. We establish this theorem by considering *maximal finitely satisfiable* sets, similarly to the proof of the compactness theorem in [35]. A set $\mathbf{S}$ of ground clauses is *finitely satisfiable*, if every finite subset of $\mathbf{S}$ is satisfiable. A finitely satisfiable set $\mathbf{S}$ is *maximal*, if no other set of ground clauses containing $\mathbf{S}$ is finitely satisfiable. Some useful properties of finitely satisfiable sets are stated below.

Lemma 9.2 *Given a finitely satisfiable set of ground clauses, $\mathbf{S}$, there exists a maximal finitely satisfiable set, $\mathbf{T}$, such that $\mathbf{S} \subseteq \mathbf{T}$.*

Proof: Let Λ be a collection of all finitely satisfiable sets of ground clauses (in a fixed language $\mathcal{L}$) that contain $\mathbf{S}$. The set Λ is partially ordered by set-inclusion. Since $\mathbf{S} \in \Lambda$, Λ is non-empty. Furthermore, for every $\subseteq$-growing chain $\Sigma \subseteq \Lambda$, the least upper bound of the chain, $\cup\Sigma$, is also in Λ. Indeed:

- $\cup\Sigma$ contains $\mathbf{S}$; and
- $\cup\Sigma$ is finitely satisfiable (for, if not, one of the elements of Σ must not be finitely satisfiable).

By Zorn's Lemma, there is a maximal element, $\mathbf{T}$, in Λ. □

Lemma 9.3 *Let $\mathbf{T}$ be a maximal finitely satisfiable set of ground clauses.*

2. The ordering, $\prec_U$, and the class membership relation, $\in_U$, are derived from the is-a assertions in $\mathbf{H}$: For all $[t], [s] \in U$, we assert $[s] \preceq_U [t]$ if and only if $s :: t \in \mathbf{H}$; and $[s] \in_U [t]$ if and only if $s : t \in \mathbf{H}$.

3. $I_{\mathcal{F}}(c) = [c]$, for every 0-ary function symbol $c \in \mathcal{F}$.

4. $I_{\mathcal{F}}(f)([t_1], ..., [t_k]) = [f(t_1, ..., t_k)]$, for every k-ary ($k \geq 1$) function symbol $f \in \mathcal{F}$.

5. $I^{(k)}_{\rightarrow}([scalM])([obj], [t_1], ..., [t_k]) = \begin{cases} [s] & \text{if } obj[scalM @ t_1, ..., t_k \rightarrow s] \in \mathbf{H} \\ \text{undefined} & \text{otherwise.} \end{cases}$

6. $I^{(k)}_{\twoheadrightarrow}([setM])([obj], [t_1], ..., [t_k]) =$

$$\begin{cases} \{[s] \mid \text{where } obj[setM @ t_1, ..., t_k \twoheadrightarrow s] \in \mathbf{H}\} & \text{if } obj[setM @ t_1, ..., t_k \twoheadrightarrow \{\}] \in \mathbf{H} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The mappings $I_{\bullet\!\!\rightarrow}$ and $I_{\bullet\!\!\twoheadrightarrow}$ are defined similarly to (5) and (6), except that inheritable data expressions must be used instead of the non-inheritable ones.

7. $I^{(k)}_{\Rightarrow}([scalM])([obj], [t_1], ..., [t_k]) =$

$$\begin{cases} \{[s] \mid \text{where } obj[scalM @ t_1, ..., t_k \Rightarrow s] \in \mathbf{H}\} & \text{if } obj[scalM @ t_1, ..., t_k \Rightarrow ()] \in \mathbf{H} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

8. $I^{(k)}_{\Rightarrow\!\!\!\Rightarrow}([setM])([obj], [t_1], ..., [t_k]) =$

$$\begin{cases} \{[s] \mid \text{where } obj[setM @ t_1, ..., t_k \Rightarrow\!\!\!\Rightarrow s] \in \mathbf{H}\} & \text{if } obj[setM @ t_1, ..., t_k \Rightarrow\!\!\!\Rightarrow ()] \in \mathbf{H} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

9. $I_{\wp}(p) = \{\langle [t_1], ..., [t_k] \rangle \mid p(t_1, ..., t_k) \in \mathbf{H}\}$.

We remark that in (6) there is a difference between the set $I^{(k)}_{\twoheadrightarrow}([setM])([obj], [t_1], ..., [t_k])$ being undefined and being empty. It is undefined if $\mathbf{H}$ contains no atoms of the form $obj[setM @ t_1, ..., t_k \twoheadrightarrow ...]$, not even $obj[setM @ t_1, ..., t_k \twoheadrightarrow \{\}]$. In contrast, $I^{(k)}_{\twoheadrightarrow}([setM])([obj], [t_1], ..., [t_k])$ is empty when $\mathbf{H}$ does contain $obj[setM @ t_1, ..., t_k \twoheadrightarrow \{\}]$, but has no atoms of the form $obj[setM @ t_1, ..., t_k \twoheadrightarrow \{s\}]$, for any $s \in U(\mathcal{F})$. Similar remarks apply to $I^{(k)}_{\Rightarrow\!\!\!\Rightarrow}$ in (7) and to $I^{(k)}_{\Rightarrow\!\!\!\Rightarrow}$ in (8).

It is easy to see that $\mathbf{I}_H = \langle U, \prec_U, \in_U, I_{\mathcal{F}}, I_{\wp}, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\!\!\rightarrow}, I_{\bullet\!\!\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rightarrow\!\!\!\Rightarrow} \rangle$ is well-defined and, indeed, is an F-structure. The above correspondence immediately leads to the following result:

Proposition 8.2 *Let* $\mathbf{S}$ *be a set of clauses. Then* $\mathbf{S}$ *is unsatisfiable if and only if* $\mathbf{S}$ *has no H-model.*

Proof: It is easy to verify that for every H-structure $\mathbf{H}$, the entailment $\mathbf{H} \models \mathbf{S}$ takes place if and only if $\mathbf{I}_H \models \mathbf{S}$, where $\mathbf{I}_H$ is the F-structure that corresponds to $\mathbf{H}$, as defined earlier. □

9 Skolemization, Clausal Form, and Herbrand's Theorem

As in classical logic, the first step in developing a resolution-based proof theory is to convert all formulas into the prenex normal form and then to *Skolemize* them. Skolemized formulas are then transformed into an equivalent *clausal form*.

Since H-structures are $\models$-closed, it is easy to see that they have *closure properties* similar to those in Section 7. These closure properties are obtained from the properties in Section 7 by replacing each $\mathbf{I} \models \phi$ there with $\phi \in \mathbf{H}$, where ϕ is a molecule, $\mathbf{I}$ is an F-structure, and $\mathbf{H}$ is an arbitrary Herbrand structure. For instance, the transitivity property for "$\doteq$":

$$\textsf{If } \mathbf{I} \models (p \doteq q) \textsf{ and } \mathbf{I} \models (q \doteq r), \textsf{ then } \mathbf{I} \models (p \doteq r)$$

now becomes:

$$\textsf{If } (p \doteq q) \in \mathbf{H} \textsf{ and } (q \doteq r) \in \mathbf{H}, \textsf{ then } (p \doteq r) \in \mathbf{H}$$

We can now define truth and logical entailment in H-structures:

Definition 8.1 (Satisfaction of Formulas by H-structures) Let $\mathbf{H}$ be an H-structure. Then:

- A ground molecule, t, is *true* in $\mathbf{H}$ (denoted $\mathbf{H} \models t$) if and only if $t \in \mathbf{H}$;
- A ground negative literal, $\neg t$, is *true* in $\mathbf{H}$ (*i.e.*, $\mathbf{H} \models \neg t$) if and only if $t \notin \mathbf{H}$;
- A ground clause, $L_1 \vee \cdots \vee L_n$, is *true* in $\mathbf{H}$ if and only if at least one literal, L_i, is true in $\mathbf{H}$;
- A clause, C, is *true* in $\mathbf{H}$ if and only if all ground instances of C are true in $\mathbf{H}$.

If every clause in $\mathbf{S}$ is true in $\mathbf{H}$, we say that $\mathbf{H}$ is a *Herbrand model* (an *H-model*) of $\mathbf{S}$. □

Correspondence between H-structures and F-structures

The above observations indicate that, as in classical logic, there should be a simple way to construct F-structures out of H-structures, and vice versa. There is one technical problem, though: Herbrand universe—the domain of all H-structures—cannot always serve as a domain of an F-structure. Indeed, in F-structures, different domain elements represent different objects. However, this is not the case with Herbrand universes. For instance, if $john \doteq father(mary)$ belongs to an H-structure, then the terms $john$ and $father(mary)$ represent the *same* object, yet they are different elements of the Herbrand universe.

The same phenomenon is encountered in classical logic with equality, and the cure for this problem is well-known. Indeed, from Section 7 and the remarks above, it follows that equality is a congruence relation on the Herbrand universe. So, we can construct a domain of an F-structure by factoring $U(\mathcal{F})$ with this relation.

The correspondence between H-structures and F-structures can now be stated as follows: Given an F-structure for a set of clauses $\mathbf{S}$, the corresponding H-structure is the set of ground F-molecules that are true in the F-structure. Conversely, for an H-structure, $\mathbf{H}$, the corresponding F-structure, $\mathbf{I}_H = \langle U, \prec_U, \in_U, I_{\mathcal{F}}, I_{\wp}, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\!\!-}, I_{\bullet\!\!\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rrightarrow} \rangle$, is defined as follows:[11]

1. The domain U is $U(\mathcal{F})/\doteq$, the quotient of $U(\mathcal{F})$ induced by the equalities in $\mathbf{H}$. We denote the equivalence class of t by $[t]$.

[11] Observe the similarity with the corresponding construction in classical logic with equality.

Trivial object-molecules:

- For every id-term, t, we have $\mathbf{I} \models t[\,]$

Constituent atoms:

- For every molecule, u, $\mathbf{I} \models u$ if and only if $\mathbf{I} \models v$ holds for every *constituent atom* v of u

The *constituent atoms* mentioned in the last property are essentially molecules broken into indivisible pieces; they are formally defined as follows:

- Every P-molecule or an is-a assertion is its own constituent atom;
- For an object-molecule, $G = P[$ *method expressions* $]$, the constituent atoms are:
  - For every signature expression $Mthd @ Q_1, ..., Q_k \twoheadrightarrow (R_1, \ldots, R_n)$ in G, where $\twoheadrightarrow$ is either $\Rightarrow$ or $\Rightarrow\!\!\!>$, the corresponding constituent atoms are:

$$P[Mthd @ Q_1, ..., Q_k \twoheadrightarrow (\,)]; \text{ and}$$
$$P[Mthd @ Q_1, ..., Q_k \twoheadrightarrow R_i\,], \quad i = 1, ..., n$$

  - For every scalar data expression $Mthd @ Q_1, ..., Q_k \rightsquigarrow S$ in G, where $\rightsquigarrow$ is either $\rightarrow$ or $\bullet\!\!\rightarrow$, the corresponding constituent atom is:

$$P[Mthd @ Q_1, ..., Q_k \rightsquigarrow S]$$

  - For every set-valued data expression, $Mthd @ Q_1, ..., Q_k \rightsquigarrow \{T_1, \ldots, T_m\}$ in G, where $\rightsquigarrow$ is either $\rightarrow\!\!\!>$ or $\bullet\!\!\rightarrow\!\!\!>$, the constituent atoms are:

$$P\,[Mthd @ Q_1, ..., Q_k \rightsquigarrow \{\,\}] \text{ and}$$
$$P\,[Mthd @ Q_1, ..., Q_k \rightsquigarrow \{T_j\}], \quad j = 1, ..., m$$

The property of scalarity follows from the simple fact that scalar invocations of methods are interpreted via single-valued functions. The other properties are straightforward from Definition 5.1.

8 Herbrand Structures

Given an F-logic language, $\mathcal{L}$, with $\mathcal{F}$ as its sets of function symbols, the *Herbrand universe* of $\mathcal{L}$ is $U(\mathcal{F})$, the set of all ground id-terms. The *Herbrand base* of $\mathcal{L}$, $\mathcal{HB}(\mathcal{F})$, is the set of all ground molecules (including P-molecules and equality).

Let $\mathbf{H}$ be a subset of $\mathcal{HB}(\mathcal{F})$; it is a *Herbrand structure* (abbr., *H-structure*) of $\mathcal{L}$ if and only if it is closed under the logical implication, "$\models$", introduced in Section 5. The requirement of $\models$-closure is needed since ground molecules may imply other molecules in a non-trivial way. For instance, $\{a :: b, b :: c\} \models a :: c$ and $\{a :: b, d[m \Rightarrow a]\} \models d[m \Rightarrow b]$. This is reminiscent of the situation in predicate calculus with equality, where sets of ground atomic formulas may imply other atomic formulas (*e.g.*, $\{a \doteq b, p(a)\} \models p(b)$).

inherited from different superclasses "adds up." For instance, suppose a class, cl, has several signature expressions for the same method, $mthd$. These signatures may be specified directly or inherited by cl. Then, "adding up" means that—as a function invoked on a member of class cl—the method $mthd$ belongs to the intersection of all functional types determined by all signatures (whether directly specified or inherited) that m has in cl.

To illustrate, consider the following example:

$empl :: person$	$person[name \Rightarrow string]$
$assistant :: student$	$student[drinks \Rightarrow\!\!\!\!\Rightarrow beer; drives \Rightarrow bargain]$
$assistant :: empl$	$empl[salary \Rightarrow int; drives \Rightarrow car]$
	$assistant[drives \Rightarrow oldThing]$

The signature accumulated by $assistant$ from all sources will then be as follows:

$$assistant[name \Rightarrow string;\ drinks \Rightarrow\!\!\!\!\Rightarrow beer;\ drives \Rightarrow (bargain, car, oldThing);\ salary \Rightarrow integer]$$

In other words, an $assistant$ inherits the type of $name$ from the class $person$, the type of $salary$ from $employee$, and his drinking habits come from the class $student$. The structure of the attribute $drives$ is determined by three factors: (i) the explicitly given signature, $drives \Rightarrow oldThing$; and (ii) the signatures inherited from the classes $student$ and $empl$. The resulting signature states that assistants drive old cars bought at bargain prices.

The next two properties of signature expressions, *input-type restriction* and *output-type relaxation*, say that when methods are viewed as functions, they have the usual properties as far as typing is concerned. For instance, the property input-type restriction says that if a method, $mthd$, returns values of type s when it is passed arguments of certain types, then $mthd$ will still return values of type s when invoked with arguments of more restricted types. Similarly, the property of output-type relaxation states that if $mthd$ returns a value of some type, r, then this value has also the type s, for any supertype s of r. Output relaxation may seem like an obvious and redundant property. However, strictly speaking, it does not logically follow from other properties.

7.4 Miscellaneous Properties

The first property in this category simply states that scalar methods are supposed to return at most one value for any given set of arguments. The second statement says that object molecules that assert no properties are trivially true in all F-structures. The last property is the *raison d'être* for the name "a molecular formula." According to this property, a molecule that is true in $\mathbf{I}$ may spin off a bunch of other, simpler, formula that are also true in $\mathbf{I}$. Moreover, these formulas cannot be decomposed further and, in a sense, they can be viewed as subformulas of the original molecule.

Scalarity:

- If $\mathbf{I} \models p[ScalM @ q_1, ..., q_k \rightarrow r_1]$ and $\mathbf{I} \models p[ScalM @ q_1, ..., q_k \rightarrow r_2]$, then $\mathbf{I} \models (r_1 \doteq r_2)$
- If $\mathbf{I} \models p[ScalM @ q_1, ..., q_k \bullet\!\!\rightarrow r_1]$ and $\mathbf{I} \models p[ScalM @ q_1, ..., q_k \bullet\!\!\rightarrow r_2]$, then $\mathbf{I} \models (r_1 \doteq r_2)$

7.2 Properties of the IS-A Relationship

The following statements say that "::" specifies a partial order on $U(\mathcal{F})$.

IS-A reflexivity:

- $\mathbf{I} \models p :: p$

IS-A transitivity:

- If $\mathbf{I} \models p :: q$ and $\mathbf{I} \models q :: r$ then $\mathbf{I} \models p :: r$

IS-A acyclicity:

- If $\mathbf{I} \models p :: q$ and $\mathbf{I} \models q :: p$ then $\mathbf{I} \models (p \doteq q)$

Subclass inclusion:

- If $\mathbf{I} \models p : q$ and $\mathbf{I} \models q :: r$ then $\mathbf{I} \models p : r$

The first three properties are direct consequences of the fact that the relation $\prec_U$ on the domain of $\mathbf{I}$ is a partial order; the last property follows from the interplay between $\prec_U$ and $\in_U$.

7.3 Properties of Signature Expressions

In the following rules, the symbol $\approx\!>$ denotes either $\Rightarrow$ or $\Rightarrow\!\!\!\!\!\Rightarrow$. Also, in the first two rules, the symbol s stands for an element of $U(\mathcal{F})$ or for "( )" — the empty conjunction of types.

Type inheritance:

- If $\mathbf{I} \models p\,[mthd \circledcirc q_1, ..., q_k \approx\!> s]$ and $\mathbf{I} \models r :: p$ then $\mathbf{I} \models r\,[mthd \circledcirc q_1, ..., q_k \approx\!> s]$

Input-type restriction:

- If $\mathbf{I} \models p\,[mthd \circledcirc q_1, ..., q_i, ..., q_k \approx\!> s]$ and $\mathbf{I} \models q_i' :: q_i$ then $\mathbf{I} \models p\,[mthd \circledcirc q_1, ..., q_i', ..., q_k \approx\!> s]$

Output-type relaxation:

- If $\mathbf{I} \models p\,[mthd \circledcirc q_1, ..., q_k \approx\!> r]$ and $\mathbf{I} \models r :: s$ then $\mathbf{I} \models p\,[mthd \circledcirc q_1, ..., q_k \approx\!> s]$

The first two properties directly follow from the anti-monotonicity constraint on the mappings $I_{\Rightarrow}(\nu(mthd))$ and $I_{\Rightarrow\!\!\!\!\!\Rightarrow}(\nu(mthd))$ in Section 5, where ν is a variable assignment. The last property follows from the upward-closedness of $I_{\Rightarrow}(\nu(mthd))(\nu(p), \nu(q_1), ..., \nu(q_k))$ and $I_{\Rightarrow\!\!\!\!\!\Rightarrow}(\nu(mthd))(\nu(p), \nu(q_1), ..., \nu(q_k))$.

Structural Inheritance and Typing

The above properties of signatures are quite interesting and merit further comments. The first of these properties, *type inheritance* (or *structural inheritance*), expresses the fact that structure propagates down from classes to subclasses. This inheritance is *monotonic* in the sense that any additional structure specified for a subclass is *added* to the structure inherited from a superclass. Moreover, any structure

- $I_{\wp}(\doteq) \stackrel{\text{def}}{=} \{\langle a, a\rangle | a \in U\}$.

This obviously implies that if T and S are id-terms, then $\mathbf{I} \models_{\nu} (T \doteq S)$ if and only if $\nu(T) = \nu(S)$.

To fully integrate P-molecules into F-logic, we now let F-formulae to be built using any kind of molecules introduced so far. Truth of formulas in F-structures and the notion of a model is defined in an obvious way.

Direct introduction of predicates serves another useful purpose: classical predicate calculus can be now viewed as a subset of F-logic. This has a very practical implication: classical logic programming, too, can be thought of as a special case of programming in F-logic. With some additional effort, most of classical logic programming theory can be adapted to F-logic, which will make it upward-compatible with the existing systems. Section 11 and Appendix A make the first step in this direction.

In this paper, we do not deal with signatures of P-molecules, as they are defined and studied in [51, 98].

7 Properties of F-structures

To get a better grip on the notions developed in the previous sections, we present a number of simple, yet important properties of F-structures. We express these properties as assertions about logical entailment, "$\models$", assertions that are true in every F-structure, $\mathbf{I}$. For simplicity, each assertion deals with ground formulas only and is an immediate consequence of the definitions; easy proofs are left as an exercise. As we shall see, many of the properties presented herein will form a basis for the inference system of F-logic developed in Section 10.

7.1 Properties of the Equality

The statements, below, express the usual properties of equality. Together, they express the fact that "$\doteq$" is a congruence relation on $U(\mathcal{F})$.

Reflexivity:

- For all $p \in U(\mathcal{F})$, $\mathbf{I} \models (p \doteq p)$

Symmetry:

- If $\mathbf{I} \models (p \doteq q)$ then $\mathbf{I} \models (q \doteq p)$

Transitivity:

- If $\mathbf{I} \models (p \doteq q)$ and $\mathbf{I} \models (q \doteq r)$ then $\mathbf{I} \models (p \doteq r)$

Substitution:

- If $\mathbf{I} \models (s \doteq t) \wedge L$, and L' is obtained by replacing an occurrence of s in L with t, then $\mathbf{I} \models L'$

with objects leads to a more natural representation. For instance, a program may mostly manipulate objects, but occasionally it may need to verify that certain objects are related via some symmetric relationship (*e.g.*, equality or adjacency). Although such relationships can always be encoded as objects, often this is not the most natural representation. Also, representing certain basic relationships, such as equality, as objects would be extremely awkward. In this section, we first show how predicates can be encoded as molecules, and then we incorporate predicate formulas into F-logic, both syntacticly and semantically.

Predicates can be simulated in several different ways. The approach described here is an adaptation from [50, 52]. To encode an n-ary predicate symbol, p, we introduce a new class for which we conveniently use the same symbol, p. Let *p-tuple* be a new n-ary function symbol. We then assert $(\forall X_1 \ldots \forall X_n)(p\text{-}tuple(X_1, ..., X_n) : p)$ and represent classical atoms of the form $p(T_1, ..., T_n)$ as molecules of the form:

$$p\text{-}tuple(T_1, ..., T_n)[arg_1 \rightarrow T_1; ...; arg_n \rightarrow T_n] \tag{2}$$

It is easily seen that the oid template, $p\text{-}tuple(T_1, ..., T_n)$, is *value-based* in the sense of [93], that is, it is fully dependent on the values of their attributes. The term $p\text{-}tuple(T_1, ..., T_n)$ can be viewed as an oid of the object to be used to represent a p-relationship among $T_1, \ldots, T_n$, if such a relationship exists; Statement (2) above *asserts* that there is an actual p-relationship among $T_1, \ldots, T_n$.

Although predicates can be represented by objects, as just described, to eliminate the need in going through the mental exercise of converting predicates into molecules, we incorporate them into our syntax and semantics directly.

The expanded language now has a set $\wp$ of *predicate symbols*. If $p \in \wp$ is an n-ary predicate symbol and T_1, ..., T_n are id-terms, then $p(T_1, ..., T_n)$ is a *predicate molecule* (abbr., P-molecule)

To avoid confusion between id-terms and P-molecules, we assume that predicates and function symbols belong to two disjoint sets of symbols, as in classical logic. In a practical logic programming language, however, this restriction may not be necessary, for, as in Prolog, it may be advantageous to overload symbols so that they could be used as predicates and as function symbols at the same time (see Section 16.2).

A (generalized) *molecular formula* is now either a molecule in the old sense, or a P-molecule (including the case of the equality predicate, *e.g.*, $T \doteq S$). A *literal* is either a molecular formula or a negated molecular formula.

Predicate symbols are interpreted as relations on U using the function $I_\wp$ such that:

- $I_\wp(p) \subseteq U^n$, for any n-ary predicate symbol $p \in \wp$.

Given an F-structure $\mathbf{I} = \langle U, \prec_U, \in_U, I_\mathcal{F}, I_\wp, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\!\!\rightarrow}, I_{\bullet\!\!\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rightarrow\!\!\!\!\Rightarrow} \rangle$ and a variable assignment ν, we write $\mathbf{I} \models_\nu p(T_1, ..., T_n)$ if and only if

- $\langle \nu(T_1), ..., \nu(T_n) \rangle \in I_\wp(p)$.

We also fix a diagonal interpretation for the equality predicate:

but $c[m@c_1, ..., c_k \Rightarrow ( )]$ is actually false, then the well-typing conditions of Section 12 mandate that $I_{\_}(\nu(m))(\nu(o), \nu(a_1), ..., \nu(a_k))$ *must* be undefined. This is analogous to a null value of the kind "value inapplicable," meaning that the value of the invocation of m on o with arguments a_1, ..., a_n does not *and cannot* exist.

- The deceptive simplicity of items (iv) and (v) is a direct result of upward-closedness of the co-domain of $I_{\Rightarrow}(\nu(ScalM))$ and $I_{\Rightarrow\Rightarrow}(\nu(SetM))$.

  Without upward-closedness, (iv) would have looked thus: $I^{(n)}_{\Rightarrow}(\nu(ScalM))(\nu(O), \nu(Q_1), ..., \nu(Q_n))$ *must be defined and, for every i, there is an element, u, in this set such that* $u \preceq_U \nu(R_i)$. A similar change would have been needed for (v).

  A more serious consequence of dumping the upward-closedness condition would be that the elegant anti-monotonicity constraint on $I_{\Rightarrow}(\nu(ScalM))$ and $I_{\Rightarrow\Rightarrow}(\nu(SetM))$ will have to be replaced with a rather awkward condition needed to coerce signatures into behaving as functional types (see Section 7.3).

- Notice *how* the above definition determines the meaning of molecules of the form $a[M @ X \rightarrow b]$ and $c[M \Rightarrow\Rightarrow d]$. The subtlety here is in how method-functions are associated with M — a variable that ranges over methods.

  If ν is a variable assignment, then $\nu(M)$ is an element in U, not in $U(\mathcal{F})$. Therefore, associating method-functions with ground id-terms is useless for interpreting the above molecules. This explains why earlier we insisted that $I_{\_}$, $I_{\Rightarrow\Rightarrow}$, etc., must be functions of the form $U \longmapsto \cdots$ and not of the form $U(\mathcal{F}) \longmapsto \cdots$.

Models and Logical Entailment

The meaning of the formulae $\varphi \vee \psi$, $\varphi \wedge \psi$, and $\neg\varphi$ is defined in the standard way: $\mathbf{I} \models_\nu \varphi \vee \psi$ (or $\mathbf{I} \models_\nu \varphi \wedge \psi$, or $\mathbf{I} \models_\nu \neg\varphi$) if and only if $\mathbf{I} \models_\nu \varphi$ or $\mathbf{I} \models_\nu \psi$ (resp., $\mathbf{I} \models_\nu \varphi$ and $\mathbf{I} \models_\nu \psi$, resp., $\mathbf{I} \not\models_\nu \varphi$). The meaning of quantifiers is also standard: $\mathbf{I} \models_\nu \psi$, where $\psi = (\forall X)\varphi$ (or $\psi = (\exists X)\varphi$), if and only if $\mathbf{I} \models_\mu \psi$ for every (resp., some) μ that agrees with ν everywhere, except possibly on X.

For a closed formula, ψ, we can omit the mention of ν and simply write $\mathbf{I} \models \psi$, since the meaning of a closed formula is independent of the choice of variable assignments.

An F-structure, $\mathbf{I}$, is a *model* of a closed formula, ψ, if and only if $\mathbf{I} \models \psi$. If $\mathbf{S}$ is a set of formulae and φ is a formula, we write $\mathbf{S} \models \varphi$ (read: φ is logically *implied* or *entailed* by $\mathbf{S}$) if and only if φ is true in every model of $\mathbf{S}$.

6 Predicates and their Semantics

It is often convenient to have usual first-order predicates on a par with objects (cf. [31, 50, 3]). This happens when an application is more naturally described in a value-based setting, or when mixing predicates

(i) $\nu(Q) \preceq_U \nu(P)$, if $G = Q :: P$; or
$\nu(Q) \in_U \nu(P)$, if $G = Q : P$.

- When G is an object molecule of the form $O[$ *method expressions* $]$ then for every method expression E in G, the following conditions must hold:

  (ii) If E is a non-inheritable scalar data expression of the form $ScalM @ Q_1, ..., Q_k \rightarrow T$, the element $I^{(k)}_{\rightarrow}(\nu(ScalM))(\nu(O), \nu(Q_1), ..., \nu(Q_k))$ must be defined and equal $\nu(T)$.

  Similar conditions must hold if E is an inheritable scalar data expressions, except that $I^{(k)}_{\rightarrow}$ should be replaced with $I^{(k)}_{\bullet\!\!\rightarrow}$.

  (iii) If E is a non-inheritable set-valued data expression, of the form $SetM @ R_1, ..., R_l \twoheadrightarrow \{S_1, ..., S_m\}$, the set $I^{(l)}_{\twoheadrightarrow}(\nu(SetM))(\nu(O), \nu(R_1), ..., \nu(R_l))$ must be defined and *contain* the set $\{\nu(S_1), ..., \nu(S_m)\}$.

  Similar conditions must hold if E is an inheritable set-valued data expression, except that $I^{(k)}_{\twoheadrightarrow}$ should be replaced with $I^{(k)}_{\bullet\!\!\twoheadrightarrow}$.

  (iv) If E is a scalar signature expression, $ScalM @ Q_1, ..., Q_n \Rightarrow (R_1, ..., R_u)$, then the set $I^{(n)}_{\Rightarrow}(\nu(ScalM))(\nu(O), \nu(Q_1), ..., \nu(Q_n))$ must be defined and contain $\{\nu(R_1), ..., \nu(R_u)\}$.

  (v) If E is a set-valued signature expression of the form $SetM @ V_1, ..., V_s \Rightarrow\!\!\!> (W_1, ..., W_v)$, the set $I^{(s)}_{\Rightarrow\!\!\!>}(\nu(SetM))(\nu(O), \nu(V_1), ..., \nu(V_s))$ must be defined and contain $\{\nu(W_1), ..., \nu(W_v)\}$. □

Here (i) says that the object $\nu(Q)$ must be a subclass or a member of the class $\nu(P)$. Conditions (ii) and (iii) say that—in case of a data expression—the interpreting function must be defined on appropriate arguments and yield results compatible with those specified by the expression. Conditions (iv) and (v) say that, for a signature expression, the type of a method ($ScalM$ or $SetM$) specified by the expression must comply with the type assigned to this method by **I**.

The following observations about Definition 5.1 are useful for better understanding the rationale behind certain aspects of the definition of F-structures:

- It follows from (iv) and (v) above that a signature of the form $c[m@c_1, ..., c_k \Rightarrow ()]$ (with nothing inside the parentheses) is *not* a tautology. Indeed, there are F-structures in which $I^{(k)}_{\Rightarrow}(\nu(m))(\nu(c), \nu(c_1), ..., \nu(c_k))$ is undefined. Similarly for $\Rightarrow\!\!\!>$.

  Such an "empty" signature intuitively says that the respective scalar method is *applicable* to objects in class c with arguments drawn from classes c_1, ..., c_k, but it does not specify the actual type of the result. Contrapositively, m cannot be applied in the above context without the signature $c[m@c_1, ..., c_k \Rightarrow ()]$ being true. In Section 12, we shall see that this enables us to enforce types by withholding signatures for certain method invocations.

- The case when $c[m@c_1, ..., c_k \Rightarrow ()]$ is true in an F-structure, **I**, but $I_{\rightarrow}(\nu(m))(\nu(o), \nu(a_1), ..., \nu(a_k))$ is *undefined*, where $\nu(o) \in_U \nu(c)$ and $\nu(a_1) \in_U \nu(a_1)$, ..., $\nu(a_k) \in_U \nu(a_k)$, is analogous to a *null value* of the kind "missing value" known from the database theory [94]. When the above value is *defined*

$v = I_{\rightarrow}^{(k)}(m)(o, \overrightarrow{args})$ is defined, it must belong to *every* class in $I_{\Rightarrow}^{(k)}(m)(cl, \overrightarrow{types})$, *i.e.*, $v \in_U w$ for each $w \in I_{\Rightarrow}^{(k)}(m)(cl, \overrightarrow{types})$. In particular, a set of types is interpreted essentially as an extended "and" of its members. This is the reason for the notation "$(\cdots)$" in the signatures (cf. Figure 4).

Similarly, the meaning of $I_{\Rightarrow\!\!\!\!\rightarrow}^{(k)}(m)$ is defined to be the type of the set-valued function $I_{\rightarrow\!\!\!\!\rightarrow}^{(k)}(m)$. In this case, since $I_{\rightarrow\!\!\!\!\rightarrow}^{(k)}(m)(o, \overrightarrow{args})$ is a *set* of objects, *every* object in $v \in I_{\rightarrow\!\!\!\!\rightarrow}^{(k)}(m)(o, \overrightarrow{args})$ must belong to *every* class, w, in $I_{\Rightarrow\!\!\!\!\rightarrow}^{(k)}(m)(cl, \overrightarrow{types})$, *i.e.*, the relationship $v \in_U w$ must hold.

The rationale behind the anti-monotonicity and the upward-closedness conditions on $I_{\Rightarrow}^{(k)}(m)$ and $I_{\Rightarrow\!\!\!\!\rightarrow}^{(k)}(m)$ now follows from the intended meaning of these functions. For instance, if the object $I_{\rightarrow}^{(k)}(m)(o, \overrightarrow{args})$ is of type cl (*i.e.*, $I_{\rightarrow}^{(k)}(m)(o, \overrightarrow{args}) \in_U cl$) then, clearly, $I_{\rightarrow}^{(k)}(m)(o, \overrightarrow{args})$ must be a member of every superclass of cl (*i.e.*, $I_{\rightarrow}^{(k)}(m)(o, \overrightarrow{args}) \in_U cl'$, for every cl' such that $cl \prec_U cl'$); thus $I_{\Rightarrow}^{(k)}(m)(cl, \overrightarrow{types})$ must be upward-closed. Similarly, if $I_{\rightarrow}^{(k)}(m)$ can be invoked on every member of class cl with proper arguments of type $\overrightarrow{types}$, then m must also be invocable on every member, o, of any subclass cl' of cl with arguments $\overrightarrow{args}$ such that $\overrightarrow{args} \in_U \overrightarrow{types}'$, where $\langle cl', \overrightarrow{types}' \rangle \preceq_U \langle cl, \overrightarrow{types} \rangle$. Furthermore, the result of this application, $v = I_{\rightarrow}^{(k)}(m)(\langle o, \overrightarrow{args} \rangle)$, must still be typed by $I_{\Rightarrow}^{(k)}(m)(cl, \overrightarrow{types})$. Now, since the type of v is given by $I_{\Rightarrow}^{(k)}(m)(cl', \overrightarrow{types}')$, it follows that $I_{\Rightarrow}^{(k)}(m)(cl', \overrightarrow{types}') \supseteq I_{\Rightarrow}^{(k)}(m)(cl, \overrightarrow{types})$, viz., anti-monotonicity.

The above relationship between $I_{\Rightarrow}$, $I_{\Rightarrow\!\!\!\!\rightarrow}$ and $I_{\rightarrow}$, $I_{\rightarrow\!\!\!\!\rightarrow}$ is not part of the definition of F-structures. Instead, it is captured at the meta-level, by the definition of type-correctness in Section 12. The relationship between $I_{\Rightarrow}$, $I_{\Rightarrow\!\!\!\!\rightarrow}$ and $I_{\bullet\!\!\rightarrow}$, $I_{\bullet\!\!\rightarrow\!\!\!\!\rightarrow}$, is also captured by the notion of type-correctness. However, it is slightly different from the relationship between $I_{\Rightarrow}$, $I_{\Rightarrow\!\!\!\!\rightarrow}$ and $I_{\rightarrow}$, $I_{\rightarrow\!\!\!\!\rightarrow}$ described above; it is fully explained in Section 12.

5.2 Satisfaction of F-formulas by F-structures

A *variable assignment*, ν, is a mapping from the set of variables, $\mathcal{V}$, to the domain U. Variable assignments extend to id-terms in the usual way: $\nu(d) = I_{\mathcal{F}}(d)$ if $d \in \mathcal{F}$ has arity 0 and, recursively, $\nu(f(..., T, ...)) = I_{\mathcal{F}}(f)(..., \nu(T), ...)$.

Molecular Satisfaction

Let $\mathbf{I}$ be an F-structure and ν a variable assignment. Intuitively, a molecule $T[\cdots]$ is *true* under $\mathbf{I}$ with respect to a variable assignment ν, denoted $\mathbf{I} \models_\nu T[\cdots]$, if and only if the object $\nu(T)$ in $\mathbf{I}$ has properties that the formula $T[\cdots]$ says it has. As a special case, molecules of the form $T[\,]$, which specify no properties, come out as tautologies. An is-a molecule, $P :: Q$ or $P : Q$, is true if the objects involved, $\nu(P)$ and $\nu(Q)$, are related via $\prec_U$ or $\in_U$ to each other.

Definition 5.1 (Satisfaction of F-Molecules) Let $\mathbf{I}$ be an F-structure and G be an F-molecule. We write $\mathbf{I} \models_\nu G$ if and only if all of the following holds:

- When G is an is-a assertion then:

Attachment of Types to Methods — The Mappings $I_{\Rightarrow}$ and $I_{\Rightarrow\Rightarrow}$

Since methods are interpreted as functions, the meaning of a signature expressions must be a functional type, for its role is to specify the type of a method. To specify a functional type, one must describe the types of the arguments to which the function can be applied and the types of the results returned by the function. Furthermore, the description should account for polymorphic types (cf. (1) earlier).

Model-theoretically, functional types are described via mappings, $I_{\Rightarrow}$ and $I_{\Rightarrow\Rightarrow}$, that cast tuples of objects to sets of objects, where objects are considered in their role as classes. These mappings must satisfy the usual properties of polymorphic functional types, as spelled out below. As in the case of $I_{\rightarrow}$ and related functions, these mappings are attached to the elements of U rather than $U(\mathcal{F})$.

We start with a very succinct definition of $I_{\Rightarrow}$ and $I_{\Rightarrow\Rightarrow}$ and then follow with a discussion of the properties of these mappings.[9]

- $I_{\Rightarrow} : U \longmapsto \prod_{i=0}^{\infty} PartialAntiMonotone_{\prec_U}(U^{i+1}, \mathcal{P}_{\uparrow}(U))$
- $I_{\Rightarrow\Rightarrow} : U \longmapsto \prod_{i=0}^{\infty} PartialAntiMonotone_{\prec_U}(U^{i+1}, \mathcal{P}_{\uparrow}(U))$

Here $\mathcal{P}_{\uparrow}(U)$ is a set of all *upward-closed* subsets of U. A set $V \subseteq U$ is *upward closed* if $v \in V$ and $v \prec_U v'$ (where $v' \in U$) imply $v' \in V$. When V is viewed as a set of classes, upward closedness simply means that, along with each class $v \in V$, this set also contains all the superclasses of v. $PartialAntiMonotone_{\prec_U}(U^{i+1}, \mathcal{P}_{\uparrow}(U))$ denotes the set of partial *anti-monotonic* functions from U^{i+1} to $\mathcal{P}_{\uparrow}(U)$. For a partial function $\rho : U^k \longmapsto \mathcal{P}_{\uparrow}(U)$, *anti-monotonicity* means that if $\vec{u}, \vec{v} \in U^k$, $\vec{v} \preceq_U \vec{u}$, and $\rho(\vec{u})$ is defined, then $\rho(\vec{v})$ is also defined and $\rho(\vec{v}) \supseteq \rho(\vec{u})$.[10]

Discussion: The Relationship between $I_{\rightarrow}$ and $I_{\Rightarrow}$

The definition of F-structures is now complete. In the rest of this subsection, we discuss the properties of $I_{\Rightarrow}$ and $I_{\Rightarrow\Rightarrow}$ and show that they coincide with the standard properties of functional types (see, *e.g.*, [25]), albeit expressed in a different, model-theoretic notation. As with $I_{\rightarrow}$, we use $I_{\Rightarrow}^{(k)}(m)$ to refer to the k-th component of the tuple $I_{\Rightarrow}(m)$. We use similar notation, $I_{\Rightarrow\Rightarrow}^{(k)}(m)$, for set-valued methods.

The intended meaning of $I_{\Rightarrow}^{(k)}(m)$ is the type of the $(k+1)$-ary function $I_{\rightarrow}^{(k)}(m)$. In other words, the domain of definition of $I_{\Rightarrow}^{(k)}(m)$ should be viewed as a set of $(k+1)$-tuples of classes, $\langle$*host-cl*, *arg-type*$_1$, ..., *arg-type*$_k\rangle$, that type tuples of arguments, $\langle o, arg_1, ..., arg_k\rangle$, on which $I_{\rightarrow}^{(k)}(m)$ can be invoked (*i.e.*, argument-tuples such that $\langle o, arg_1, ..., arg_k\rangle \in_U \langle$*host-cl*, *arg-type*$_1$, ..., *arg-type*$_k\rangle$ holds). For every tuple of classes, $\langle cl, \overrightarrow{types}\rangle \in U^{k+1}$, if $I_{\Rightarrow}^{(k)}(m)(cl, \overrightarrow{types})$ is defined, it represents the type of $I_{\rightarrow}^{(k)}(m)(o, \overrightarrow{arg}s)$ for any tuple of arguments such that $\langle o, \overrightarrow{arg}s\rangle \in_U \langle cl, \overrightarrow{types}\rangle$. This means that if

[9] Signatures constitute a fairly advanced level of F-logic; its *basic* features—is-a hierarchy and data expressions—do not depend on the specifics of the above type system. For this reason, on the first reading it may be possible to skip signature-related aspects of F-logic, including the remaining part of this subsection.

[10] Actually, these functions are *monotone* with respect to *Smyth's ordering* [24]. For upward-closed sets, $S \subseteq^{smyth} S'$ if and only if $S \supseteq S'$.

whether the method is invoked as a scalar or a set-valued function. Therefore, to assign meaning to methods, an F-structure has to attach an appropriate function to each method.

As explained in Section 3, F-logic allows any id-term to be used as a method name. Hence, we need to associate a function to each id-term. However, to provide for higher-order jobs, such as schema manipulation, one has to allow variables to range over methods. It turns out that to give meaning to molecules with method-variables, it is necessary to associate these functions with each element of U, not $U(\mathcal{F})$.[8] Furthermore, since methods can have different arities, we need to associate a function for each possible arity.

Formally, in their role as methods, objects are interpreted via an assignment of appropriate functions to each element of U, using the maps $I_{\rightarrow}$, $I_{\bullet\!\!\rightarrow}$, $I_{\twoheadrightarrow}$, and $I_{\bullet\!\!\twoheadrightarrow}$. More precisely, for every object, its incarnation as a scalar method is obtained via the mapping

- $I_{\rightarrow}, I_{\bullet\!\!\rightarrow} : U \longmapsto \prod_{k=0}^{\infty} Partial(U^{k+1}, U)$

Each of these mappings associates a tuple of partial functions $U^{k+1} \rightarrow U$ with every element of U; there is exactly one such function in the tuple, for every method-arity $k \geq 0$. In other words, the same method can be invoked with different arities.

In addition to different arities, every method can be invoked as a scalar or as a set-valued function (cf. $empl[jointWorks@bill \twoheadrightarrow X]$ and $student[jointWorks@bill \rightarrow X]$ in Section 3). Semantically this is achieved by interpreting the set-valued incarnations of methods separately, via the mappings:

- $I_{\twoheadrightarrow}, I_{\bullet\!\!\twoheadrightarrow} : U \longmapsto \prod_{k=0}^{\infty} Partial(U^{k+1}, \mathcal{P}(U))$

For every method-arity k, each of these mappings associates a partial function $U^{k+1} \rightarrow \mathcal{P}(U)$ with each element of U. Note that each element of U has four different sets of interpretations: two provided by $I_{\rightarrow}$ and $I_{\bullet\!\!\rightarrow}$ and two provided by $I_{\twoheadrightarrow}$ and $I_{\bullet\!\!\twoheadrightarrow}$.

The difference between the "$\rightarrow$"-versions and the "$\bullet\!\!\rightarrow$"-versions in the above mappings is that "$\rightarrow$"-versions are used to interpret inheritable data properties, while "$\bullet\!\!\rightarrow$"-versions are for non-inheritable data properties.

As seen from the above definitions, $I_{\rightarrow}(m)$ (and $I_{\twoheadrightarrow}(m)$, $I_{\bullet\!\!\rightarrow}(m)$, $I_{\bullet\!\!\twoheadrightarrow}(m)$), where $m \in U$, is an infinite tuple of functions parameterized by the arity $k \geq 0$. To refer to the k-th component of such a tuple, we use the notation $I_{\rightarrow}^{(k)}(m)$ (resp., $I_{\twoheadrightarrow}^{(k)}(m)$, $I_{\bullet\!\!\rightarrow}(m)$ or $I_{\bullet\!\!\twoheadrightarrow}(m)$). Thus, a method, m, that occurs in a scalar non-inheritable data expression with k proper arguments is interpreted by $I_{\rightarrow}^{(k)}(m)$; if m occurs in a set-valued non-inheritable data expression with k arguments, it is interpreted by $I_{\twoheadrightarrow}^{(k)}(m)$, and so on. Note that $I_{\rightarrow}^{(k)}(m)$ and the other three mappings are $(k+1)$-ary functions. The first argument here is the host object of the invocation of the method m; the other k arguments correspond to the *proper* arguments of the invocation. In the parlance of object-oriented systems, an expression, such as $I_{\rightarrow}^{(k)}(m)(obj, a_1, ..., a_k)$, is interpreted as a request to object obj to invoke a scalar method, m, on the arguments $a_1, ..., a_k$.

[8] This is a subtle technical point whose necessity will become apparent after the notion of satisfaction in F-structures is introduced.

5 Semantics

Given a pair of sets U and V, we shall use $Total(U,V)$ to denote the set of all total functions $U \longmapsto V$; similarly, $Partial(U,V)$ stands for the set of all partial functions $U \longmapsto V$. The power-set of U will be denoted by $\mathcal{P}(U)$. Further, given a collection of sets $\{S_i\}_{i\in\mathcal{N}}$ parameterized by natural numbers, $\prod_{i=1}^{\infty} S_i$ will denote the Cartesian product of the S_i's, that is, the set of all infinite tuples $\langle s_1, \ldots, s_n, \ldots\rangle$.

5.1 F-structures

In F-logic, semantic structures are called *F-structures*. Given a language of F-logic, $\mathcal{L}$, an *F-structure* is a tuple $\mathbf{I} = \langle U, \prec_U, \in_U, I_{\mathcal{F}}, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\!\!\rightarrow}, I_{\bullet\!\!\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rightarrow\!\!\!\!\Rightarrow}\rangle$. Here U is the domain of $\mathbf{I}$, $\prec_U$ is a partial order on U, and $\in_U$ is a binary relation. As usual, we write $a \preceq_U b$ whenever $a \prec_U b$ or $a = b$. We extend $\preceq_U$ and $\in_U$ to tuples over U in a natural way: for $\vec{u}, \vec{v} \in U^n$ and $S \subseteq U^n$, we write $\vec{u} \preceq_U \vec{v}$ or $\vec{u} \in_U \vec{v}$ if the corresponding relationships hold between $\vec{u}$ and $\vec{v}$ component-wise.

The ordering $\prec_U$ on U is a semantic counterpart of the subclass relationship, *i.e.*, $a \prec_U b$ is interpreted as a statement that a is a subclass of b. The binary relation $\in_U$ will be used to model class membership, *i.e.*, $a \in_U b$ should be taken to mean that a is a member of class b. The two binary relationships, $\prec_U$ and $\in_U$, are related as follows: If $a \in_U b$ and $b \preceq_U c$ then $a \in_U c$. This is just another way of saying that the extension of a subclass (*i.e.*, its set of members) is a subset of the extension of a superclass.

We do not impose any other restrictions on the class membership relation to accommodate the widest range of applications. In particular, $\in_U$ does not have to be acyclic and even $s \in_U s$ is a possibility (*i.e.*, a class may be a member of itself when viewed as an object).[7] The reader should not be misled into thinking that v in $u \in_U v$ is a subset of U that contains u. The actual meaning of such a statement is that v is an element of U that *denotes* a subset of U, and u is a member of this subset.

By analogy with classical logic, we can view U as a set of all *actual* objects in a *possible world*, $\mathbf{I}$. Ground id-terms (the elements of $U(\mathcal{F})$) play the role of logical object id's. They are interpreted by the objects in U via the mapping $I_{\mathcal{F}} : \mathcal{F} \longmapsto \cup_{i=0}^{\infty} Total(U^i, U)$. This mapping interprets each k-ary object constructor, $f \in \mathcal{F}$, by a function $U^k \longmapsto U$. For $k = 0$, $I_{\mathcal{F}}(f)$ can be identified with an element of U. Thus, function symbols are interpreted the same way as in predicate calculus.

The remaining six symbols in $\mathbf{I}$ denote mappings for interpreting each of the six types of method expressions in F-logic. These mappings are described next.

Attachment of Functions to Methods — The Mappings $I_{\rightarrow}$, $I_{\twoheadrightarrow}$, $I_{\bullet\!\!\rightarrow}$, and $I_{\bullet\!\!\twoheadrightarrow}$

As in classical logic, the mapping $I_{\mathcal{F}}$ above is used to associate an element of U with each id-term. However, id-terms can also be used to denote methods. A method is a function that takes a host object and a list of proper arguments and maps them into another object or a set of objects, depending on

[7] Such flexibility is sometimes required in AI applications, where s would be interpreted as a "typical" element of class s.

It follows from the syntax that every logical id can denote an entity or a method, depending on the syntactic position of this id within the formula. In an occurrence as a method, this id can denote either a *scalar* function or a *set-valued* function. The type of the invocation (scalar or set-valued) is determined by the context, *viz.*, by the type of the associated arrow.

Is-a assertions are always atomic—they cannot be decomposed further into simpler formulas. Other molecular formulas, however, are not always atomic. As will be seen shortly, a molecule such as $X[attr_1 \rightarrow a; attr_2 \rightarrow Y]$ is equivalent to a conjunction of its *atoms*, $X[attr_1 \rightarrow a] \wedge X[attr_2 \rightarrow Y]$. It is for this property that we call such formulas "molecular" instead of "atomic." Atomic formulas will be defined later, in Section 7.

Complex Formulas

F-formulae are built up from simpler F-formulae by means of logical connectives and quantifiers:

- Molecular formulae are F-formulas;
- $\varphi \vee \psi$, $\varphi \wedge \psi$, $\neg\varphi$ are F-formulae, if so are φ and ψ;
- $\forall X\, \varphi$, $\exists Y\, \psi$ are formulae, if so are φ, ψ and X, Y are variables.

In addition, we define a *literal* to be either a molecular formula or a negation of a molecular formula.

In Section 3 and elsewhere in this paper, we shall often use the *implication* connective, " $\longleftarrow$ ". In F-logic, this connective is defined as in classical logic: $\varphi \longleftarrow \psi$ is just another way of saying $\varphi \vee \neg\psi$. There is a tradition to refer to logical statements written in the implicative form as *rules*. This terminology was already used in the example of Section 3, and we shall continue this practice.

It is sometimes convenient to combine different kinds of molecules (as in (vii), (viii) and (ix) of Figure 4) and write, say,

$$Q : P[ScalM @ X \rightarrow (Y : S);\ SetM @ Y, W \twoheadrightarrow (Z : R, T)]$$

as a short-hand for

$$Q : P \wedge Q[ScalM @ X \rightarrow Y] \wedge Y : S \wedge Q[SetM @ Y, W \twoheadrightarrow (Z, T)] \wedge Z : R.$$

Furthermore, even though the symbols on the right-hand side of the arrows denote id-terms by definition, it is often convenient (and, in fact, customary) to combine molecules as in (i) and (ii) of Figure 4. For instance,

$$P[ScalM @ X \rightarrow Q[SetM @ Y \twoheadrightarrow \{T, S\}]]$$

can be defined as a short-hand for

$$P[ScalM @ X \rightarrow Q] \wedge Q[SetM @ Y \twoheadrightarrow \{T, S\}]$$

In data expressions, the terms T and S_i are id-terms that represent output of the respective methods, $ScalarMethod$ and $SetMethod$ when they are invoked on the host-object O with the arguments Q_1, ..., Q_k and R_1, ..., R_l, respectively. The arguments are id-terms.

In signature expressions, A_i and B_j are id-terms that represent *types* of the results returned by the respective methods when invoked on an object of class C with arguments of types V_1, ..., V_n and W_1, ..., W_s, respectively; these arguments are also id-terms. The notation $(\cdots)$ in signature expressions is intended to say that the output of the method must belong to *all* the classes listed to the right of "$\Rightarrow$" or "$\Rrightarrow$".

The order of data and signature expressions in a molecule is immaterial. For convenience, the same method and even the same data/signature expression may have multiple occurrences in the same molecule. Likewise, the same id-term may occur multiple times inside the braces in a data, or signature expression. Furthermore, whenever a method does not expect arguments, "@" will be omitted. For instance, we will write $P[Mthd \rightarrow Val]$ instead of $P[Mthd@ \rightarrow Val]$, and similarly for $\twoheadrightarrow$, $\Rightarrow$, and $\Rrightarrow$. Likewise, when only one element appears inside { }, we may write $P[\ldots \twoheadrightarrow S]$ instead of $P[\ldots \twoheadrightarrow \{S\}]$; we shall also write $Q[\ldots \Rightarrow T]$ and $Q[\ldots \Rrightarrow T]$ instead of $Q[\ldots \Rightarrow (T)]$, and $Q[\ldots \Rrightarrow (T)]$.

Discussion

Informally, an object molecule in (ii) above asserts that the object denoted by O has properties specified by the method expressions. Data expressions are used to define properties of objects in terms of what their methods are supposed to do. Inheritable data expressions may be inherited by subclasses and individual members of the object (when it plays the role of a class); by contrast, properties specified as non-inheritable cannot be inherited. A signature expression in (ii) specifies type constraints on the methods applicable to the objects in class O. Typing is given both for method arguments and for their results. For instance, the scalar signature expression in (ii) says that $ScalarMethod$ is a scalar method and that when it is invoked on a host-object of class O with proper arguments coming from classes V_1, ..., V_n, then the result must simultaneously belong to classes A_1, ..., A_r. Similarly, the typing for $SetMethod$ in (ii) says that it is a set-valued method; when it is invoked on a host object of class O with proper arguments of classes W_1, ..., W_s, then *each* element in the resulting set must simultaneously belong to the classes B_1, ..., B_t.

Notice that a molecule, such as $a[attr \rightarrow b; attr \twoheadrightarrow \{c,d\}; attr \bullet\!\!\rightarrow e]$, is syntactically well-formed despite the fact that *attr* is used to specify a non-inheritable scalar property of the object a due to one part of the molecule, a non-inheritable set-valued property due to another, and also an inheritable scalar property due to the third part. This apparent contradiction is easily resolved at the semantic level: The attribute *attr* has value b on object a when invoked as a non-inheritable scalar method (with the arrow "$\rightarrow$"); it returns the set $\{c,d\}$ when called as a non-inheritable set-valued method (with "$\twoheadrightarrow$"); and it returns the value e when called as an inheritable scalar method (with the arrow "$\bullet\!\!\rightarrow$"). If a happens to be a class-object, the properties $attr \rightarrow b$ and $attr \twoheadrightarrow \{c,d\}$ are not inheritable by the members and subclasses of a. However, $attr \bullet\!\!\rightarrow e$ is inheritable, as it is specified as such.

In this paper, we adopt a convention inspired by Prolog syntax, whereby a symbol that begins with a lower-case letter denotes a ground id-term and a symbol that begins with a capital letter denotes an id-term that may be non-ground.

Definition 4.1 (Molecular Formulas) A *molecule* in F-logic is one of the following statements:

(i) An *is-a assertion* of the form $C :: D$ or of the form $O : C$, where C, D, and O are id-terms.

(ii) An *object molecule* of the form O [semicolon-separated list of *method expressions*].

A *method expression* can be either a *non-inheritable data expression*, an *inheritable data expression*, or a *signature expression*.

- *Non-inheritable data expressions* take one of the following two forms:
  - A non-inheritable *scalar* expression ($k \geq 0$):

$$ScalarMethod@Q_1, ..., Q_k \rightarrow T$$

  - A non-inheritable *set-valued* expression ($l, m \geq 0$):

$$SetMethod@R_1, ..., R_l \twoheadrightarrow \{S_1, ..., S_m\}$$

- *Inheritable* scalar and set-valued data expressions are like non-inheritable expressions except that "$\rightarrow$" is replaced with "$\bullet\!\!\rightarrow$" and "$\twoheadrightarrow$" is replaced with $\bullet\!\!\twoheadrightarrow$.
- *Signature expressions* also take two forms:
  - A *scalar* signature expression ($n, r \geq 0$):

$$ScalarMethod@V_1, ..., V_n \Rightarrow (A_1, \ldots, A_r)$$

  - A *set-valued* signature expression ($s, t \geq 0$):

$$SetMethod@W_1, ..., W_s \Rightarrow\!\!\!\Rightarrow (B_1, \ldots, B_t)$$

□

The first is-a assertion in (i), $C :: D$, states that C is a *nonstrict* subclass of D (*i.e.*, inclusive of the case when C and D denote the same class).[6] The second assertion, $O : C$, states that O is a member of class C.

In (ii), O is an id-term that denotes an object. $ScalarMethod$ and $SetMethod$ are also id-terms. However, the syntactic position of $ScalarMethod$ indicates that it is invoked on O as a scalar method, while the syntactic position of $SetMethod$ indicates a set-valued invocation. (If $ScalarMethod$ and $SetMethod$ have variables, each of these terms denotes a family of methods rather than a single method.) Double-headed arrows, $\twoheadrightarrow$, $\bullet\!\!\twoheadrightarrow$, and $\Rightarrow\!\!\!\Rightarrow$, indicate that $SetMethod$ denotes a set-valued function. The single-headed arrows, $\rightarrow$, $\bullet\!\!\rightarrow$, and $\Rightarrow$, indicate that the corresponding method is scalar.

[6] Assertions, such as $person :: person$, will later turn out to be tautologies, *i.e.*, any class is a subclass of itself.

Despite the higher-order syntax, the underlying semantics of F-logic formally remains first-order,[5] which explains why it is possible to circumvent difficulties normally associated with higher-order theories. These important issues are beyond the scope of this paper; a more complete discussion appears in [30].

4 Syntax

The *alphabet* of an F-logic language, $\mathcal{L}$, consists of:

- a set of *object constructors* $\mathcal{F}$;
- an infinite set of *variables* $\mathcal{V}$;
- auxiliary symbols, such as, (,), [,], $\rightarrow$, $\twoheadrightarrow$, $\bullet\!\!\rightarrow$, $\bullet\!\!\twoheadrightarrow$, $\Rightarrow$, $\Rightarrow\!\!>$, etc; and
- usual logical connectives and quantifiers, $\vee$, $\wedge$, $\neg$, $\longleftarrow$, $\forall$, $\exists$.

Object constructors (elements of $\mathcal{F}$) play the role of function symbols of F-logic. Each function symbol has an *arity*—a nonnegative integer that determines how many arguments this symbol can take. Symbols of arity 0 play the role of constant symbols; symbols of arity ≥ 1 are used to construct larger terms out of simpler ones. An *id-term* is a usual first-order term composed of function symbols and variables, as in predicate calculus. The set of all *ground* (*i.e.*, variable-free) id-terms is denoted by $U(\mathcal{F})$. This set is also commonly known as *Herbrand Universe*.

Conceptually, ground id-terms play the role of *logical object id's*—a logical abstraction of the implementational concept of *physical object identity* [46, 3]. Since this paper emphasizes logic, the term "object id" (abbr., *oid*) will be used for logical id's only.

Objects represented by "complex" id-terms, such as $addr(13, mainstreet, anywhere)$, usually arise when a complex object (or a class) is constructed out of simpler components, *e.g.*, 13, *mainstreet*, and *anywhere*, in this example.

It follows from the definition that every F-logic language is uniquely determined by its set of object constructors, $\mathcal{F}$. In this paper, $\mathcal{F}$ and $U(\mathcal{F})$ will henceforth denote the set of function symbols and ground terms, respectively, where the language, $\mathcal{L}$, will be known from the context.

Molecular Formulas

A language of F-logic consists of a set of formulae constructed out of the alphabet symbols. As in most other logics, formulas are built out of simpler formulas by using the usual connectives $\neg$, $\vee$, and $\wedge$, and quantifiers $\exists$ and $\forall$. The simplest kind of formulas are called *molecular* F-formulas (abbr., *F-molecules* or just *molecules*).

[5] Following [30], first-orderness here means that variables do not range over complex domains, such as domains of sets or functions, but they can range over the intensions of those higher-order structures.

say that *plus* is a method that returns an integer when invoked on an *integer*-object with an *integer*-argument. However, when *plus* is invoked on an object in class *real* with an argument that also comes from that class, the method returns an object in class *real*. In general, just as in (1), specifying polymorphic types requires more than one signature. Several interesting examples of polymorphic types (including parameterized types) will be given in Section 11.3.

A form of polymorphism when a method can be invoked with varying number of arguments is called *arity-polymorphism*. For instance, $student[avgGrade \Rightarrow grade]$ and $student[avgGrade@year \Rightarrow grade]$ may both be legal invocations (say, the first implicitly assuming the current year).

Arity-polymorphism is very popular in logic programming languages, such as Prolog. However, unlike Prolog, arity polymorphism in F-logic is controlled via signatures, just as any other kind of polymorphism. This means that to be able to invoke a method with any given number of arguments, an appropriate signature must be specified. For instance, in the above, if no other signature is given, *avgGrade* cannot be invoked with more than one argument. Another way to control arity-polymorphism is by turning F-logic into a sorted language. This extension is described in Section 16.

One more kind of polymorphism arises when a method name is declared to be both set-valued and scalar. For instance, suppose the following types are defined:

$$student[grade@course \Rightarrow\!\!\!> integer] \qquad\qquad student[grade@course \Rightarrow integer]$$

In the first case, *grade* is a set-valued method of one argument that for any student and any given course returns the set of this student's scores in the course (say, the scores for all projects and examinations). In the second case, only the overall grade for the course is returned. For instance, if *sally* is a student, the query

$$?- \quad sally[grade@db \rightarrow\!\!\!> X]$$

will return the set of Sally's scores in the database course, while the query

$$?- \quad sally[grade@db \rightarrow X]$$

will return the final grade. This kind of polymorphism is controlled as before, via signatures; it can also be controlled via sorts, similarly to arity-polymorphism.

Observe that F-logic manipulates several higher-order concepts. In Figure 4, the attribute *friends* is a set-valued function, *i.e.*, it returns sets of objects. Similarly, the user can think of the class-objects in the IS-A hierarchy of Figure 2 as sets ordered by the subset relation.

Furthermore, attributes and methods are also viewed as objects. This, for instance, implies that their names can be returned as query answers. In this way, schema information is turned into data so that it can be manipulated in the same language. This is useful for tasks that require schema exploration in databases (Section 11.4.2), inheritance with overriding (Section 14.3), and for many other applications (Sections 11.4.3 and 11.4.5).

Other Features

Before bidding farewell to this example, we would like to highlight some other features of F-logic. Suppose that in Statement (i) we replace $mngr \rightarrow bob$ with $mngr \rightarrow phil$. Then we would have had a *type error*. Indeed, on the one hand, the deductive rule (vii) implies $bob[boss \rightarrow phil]$; on the other hand, from Figure 2, *phil* is neither a faculty nor a manager. This contradicts the typing of the attribute *boss* in (iii).[3] As will be seen in Section 12, the notions of well-typing and type error have precise *model-theoretic* meaning in F-logic.

Another aspect of the type system of F-logic is that signature declarations are enforced. For instance, the following methods are declared in Figure 4 for the members of class *faculty*:

- *name*, *friends*, and *children*—the methods inherited from *person*;
- *worksFor* and *jointWorks*—methods inherited from *empl*; and
- *boss*, *age*, *highestDegree*, and *papers*—the methods directly specified for the class *faculty*.

Enforcing signature declarations means that these are the *only* methods applicable to the members of class *faculty*; any other method is illegal in the scope of that class. Moreover, it is a type error to invoke the method *jointWorks* with an argument that is not a member of class *empl*. Similarly, it would be a type error to invoke the methods in the last group on objects that represent employees who are not faculty, or to invoke *worksFor* or *jointWorks* on objects that are not members of class *empl*.

In our example, invocations of all methods, except *avgSalary*, are sanctioned by signatures declared for appropriate classes. The method *avgSalary*, on the other hand, is not covered by any signature, which is a violation of the well-typing conditions (to be discussed in Section 12). To correct the problem, the class *faculty* (in its role as an object) has to be made into a member of another class, say *employmentGroup* (whose members are various categories of employees, such as *empl*, *faculty*, and *manager*). To give *avgSalary* a proper type, we could then declare $employmentGroup[avgSalary \Rightarrow integer]$.

Observe a difference in the treatment of the non-inheritable method *avgSalary* (and of all other non-inheritable methods in the example) and of the inheritable method *highestDegree*. The former must be covered by a signature declared for a class where *faculty* is a *member*, while the latter should be covered by a signature declared for a class where *faculty* is a *subclass*.[4] This is because the non-inheritable method *avgSalary* is a property of the object *faculty* itself, while the inheritable method *highestDegree* is effectively a property of the objects that are members of the class *faculty*.

Yet another important aspect of F-logic type system is *polymorphism*. This means that methods can be invoked on different classes with arguments of different types. For instance, the following signatures

$$integer[plus@integer \Rightarrow integer] \qquad real[plus@real \Rightarrow real] \tag{1}$$

[3] Note that *phil* is an employee and thus $bob[boss \rightarrow phil]$ complies with the typing of *boss* for the class *empl*, as specified in (v). However, *bob* is also a member of class *faculty*, and (iii) imposes a stricter type on the attribute *boss* in that class, requiring *phil* to be a member of both *faculty* and *manager*.

[4] This class can be *faculty* itself, since the subclass relationship in F-logic is non-strict.

$friends \twoheadrightarrow \{bob\}$. However, in $friends \twoheadrightarrow \{bob, sally\}$ of Statement (ii) and in $assistants \twoheadrightarrow \{john, sally\}$ in (i), braces must be kept to indicate sets. (In fact, say, $cs_1[assistants \twoheadrightarrow \{john, sally\}]$ is equivalent to a conjunction of $cs_1[assistants \twoheadrightarrow john]$ and $cs_1[assistants \twoheadrightarrow sally]$, but the use of braces leads to a shorter notation).

Statement (vii) is a deductive rule that defines a new attribute, *boss*, for objects of class *empl*. It says that an employee's boss is the manager of the department the employee works in. Here we follow the standard convention in logic programming that requires names of logical variables to begin with a capital letter. Statement (viii) defines a method $jointWorks$ whose signature is given in (v). For any object in class *person*, $jointWorks$ is a function that takes an object of type *person* and returns a set of objects of type *report*; each object in this set represents a paper co-authored by the two people. Informally, this rule should be read as follows: If a *report*-object, Z, simultaneously belongs to the sets $X.papers$ and $Y.papers$,[2] where X and Y are *faculty*-objects, then Z must also belong to the set returned by the method $jointWorks$ when invoked on the host-object X with the argument Y.

It is important to realize here that the variable Z ranges over the *members* of the set $X.papers \cap Y.papers$—it is *not* instantiated to the set itself. Note also that we do not need the restriction $Z : report$ in the rule body because, for each $faculty$, the attribute *papers* specifies a set of $article$-objects (by Statement (iii)), and $article$ is a subclass of *report* (see Figure 2). However, if we were interested in JACM papers only, then the restriction $Z : jacm$ would have been necessary in the body of (viii).

Two uses of the method $jointWorks$ are shown in (x) and (xi). Query (x) asks about the co-authors of *mary*'s paper, $jacm90$, while (xi) requests all joint papers that *mary* co-authors with *phil*. Query (ix) inquires about all middle-aged employees working for "CS" departments. In particular, for every such employee, the attributes *boss*, *age*, and *worksFor* are requested. The expected answer to (ix) is:

(xii) $bob[boss \rightarrow bob; age \rightarrow 40; worksFor \rightarrow cs_1]$.

The object *mary* does not qualify as an answer to query (ix) because of the unknown age.

In this connection, we would like to mention the fact that attributes and methods are *partial* functions. They may be defined on some objects in a class and undefined on another. F-logic distinguishes two reasons for undefinedness: 1) the attribute (or method) may be *inapplicable* to the object; or 2) it may be applicable but its value is unknown. Case 1) arises due to typing, as discussed later. Case 2), on the other hand, is essentially a form of a *null value*, which is familiar from database theory.

In the above example and in the rest of this paper, the term "type" of an object is used interchangeably to refer to classes of that object and to its declared signatures. The correct meaning should be clear from the context. The dual use of this term is appropriate since, generally, the term "type" refers to arbitrary collections of abstract values. Classes and signatures both specify such collections. The former denotes collections of class members, which are semantically related objects; the latter denotes collections of objects that are structurally related. Since semantic similarity usually implies structural similarity, the two uses of the term "type" are closely related.

[2]Here the notation $X.papers$ denotes the set of oid's returned by the attribute *papers* on the object X. Similarly for $Y.papers$.

in $cl[attr \Rightarrow (a, b, c)]$ signifies a conjunction of types: the value of *attr* on any object in class *cl* must simultaneously belong to each of the classes *a*, *b*, and *c*. When the set of types to the right of a double-shafted arrow, "$\Rightarrow$" or "$\Rightarrow\!\!\!>$", is a singleton set, *e.g.*, $c[attr \Rightarrow (empl)]$, we shall omit the surrounding parentheses and write, *e.g.*, $c[attr \Rightarrow empl]$.

Different kinds of information about objects can be mixed in one statement. For instance, in Statement (iii), the expression $highestDegree \Rightarrow degree$ specifies the type of the attribute *highestDegree*; the expression $highestDegree \bullet\!\!\rightarrow phd$ specifies an *inheritable property* of *faculty*; and $avgSalary \rightarrow 50000$ is a *non-inheritable property* of *faculty*.

Asserting an inheritable property for a class-object has the effect that every member-object of that class inherits this property, unless it is overwritten. For instance, the assertion $bob[highestDegree \rightarrow phd]$ is derivable from (iii) by inheritance. Note that when a property is inherited by a member of the class, it becomes *non-inheritable*; this explains why inheritance derives $bob[highestDegree \rightarrow phd]$ rather than $bob[highestDegree \bullet\!\!\rightarrow phd]$. An inheritable property may also be inherited by a subclass. However, in this case, the inherited property remains inheritable in this subclass and, as such, it can be passed further down in the hierarchy of objects. For example, if *lecturer* were a subclass of *faculty* then $lecturer[highestDegree \bullet\!\!\rightarrow phd]$ would be derivable by inheritance, unless *lecturer* has another inheritable property (*e.g.*, $highestDegree \bullet\!\!\rightarrow ms$) that overrides this inheritance. Inheritance will be discussed in detail in Section 14.

In contrast to $highestDegree \bullet\!\!\rightarrow phd$, the property $avgSalary \rightarrow 50000$ in (iii) is not inheritable by the members and subclasses of *faculty*. And, indeed, it makes no sense to inherit average salary, as it is an aggregate property of all members of the class and has no meaning for individual members. Inheriting $avgSalary \rightarrow 50000$ to a subclass is also meaningless, because subclass members, *e.g.*, all lecturers, are likely to have a different average salary than all members of the larger class *faculty*.

Statements (iv) to (vi) specify typing constraints for classes *person*, *empl*, and *dept*. More precisely, it can be said that these statements define *signatures* of methods attached to these classes. Two things are worth noting in Statement (iv). First, the expression $jointWorks@person \Rightarrow\!\!\!> report$ describes a method, *jointWorks*, that expects one proper argument of type *person* and returns a set of elements of type *report*. In object-oriented terms, this means that whenever a *person*-object, *obj*, receives a message, *jointWorks*, with an argument of class *person*, then the reply-message returned by *obj* will consist of a set of objects, each being a member of class *report*. Note that there is no essential difference between methods and attributes: the latter are simply methods without arguments. Strictly speaking, in Figure 4 we should have written $name@ \Rightarrow string$ and $age@ \rightarrow 40$ instead of $name \Rightarrow string$ and $age \rightarrow 40$, but the latter short-hand notation is more convenient when no arguments are expected.

The second thing to note in (iv) is the expression $children \Rightarrow child(person)$, which specifies a type constraint for the attribute *children*. Here, *child* is a unary function symbol and *person* is a constant denoting a class. The term $child(person)$, then, is a logical id of another class. Thus, in F-logic, function symbols are used as constructors of object and class id's.

We shall often omit braces surrounding singleton sets and write, say, $friends \rightarrow\!\!\!> bob$ instead of

Database Facts:

(i) $bob[name \rightarrow \text{“}Bob\text{”}; age \rightarrow 40;$
$worksFor \rightarrow cs_1[dname \rightarrow \text{“}CS\text{”}; mngr \rightarrow bob; assistants \twoheadrightarrow \{john, sally\}]\,]$

(ii) $mary[name \rightarrow \text{“}Mary\text{”}; highestDegree \rightarrow ms;$
$friends \twoheadrightarrow \{bob, sally\}; worksFor \rightarrow cs_2[dname \rightarrow \text{“}CS\text{”}\,]\,]$

General Class Information:

(iii) $faculty[boss \Rightarrow (faculty, manager);$ % Typing: *boss*—scalar attribute; returns objects
$age \Rightarrow midaged; highestDegree \Rightarrow degree;$ % belonging to both *faculty* & *manager*
$papers \Rightarrow\!\!> article; highestDegree \bullet\!\!\rightarrow phd;$ % $highestDegree \bullet\!\!\rightarrow phd$: inheritable property
$avgSalary \rightarrow 50000\,]$ % $avgSalary \rightarrow 50K$: non-inheritable property

(iv) $person[name \Rightarrow string; friends \Rightarrow\!\!> person;$ % Typing: *friends*—a set-valued attribute
$children \Rightarrow\!\!> child(person)]$ % Typing: each child is in class $child(person)$

(v) $empl[worksFor \Rightarrow dept; boss \Rightarrow empl;$ % Typing: employees work for departments
$jointWorks@empl \Rightarrow\!\!> report\,]$ % Typing: *jointWorks*—a one-argument method

(vi) $dept[assistants \Rightarrow\!\!> (student, empl); mngr \Rightarrow empl\,]$ % Typing: assistants are students and employees

Deductive Rules:

(vii) $E[boss \rightarrow M\,] \leftarrow E : empl \wedge D : dept \wedge E[worksFor \rightarrow D[mngr \rightarrow M : empl\,]\,]$

(viii) $X[jointWorks@Y \twoheadrightarrow Z\,] \leftarrow Y : faculty \wedge X : faculty$
$\wedge\, Y[papers \twoheadrightarrow Z\,] \wedge X[papers \twoheadrightarrow Z\,]$

Queries:

(ix) $?- \;\; X : empl \wedge X[boss \rightarrow Y; age \rightarrow Z : midaged; worksFor \rightarrow D[dname \rightarrow \text{“}CS\text{”}\,]\,]$

(x) $?- \;\; mary[jointWorks@Y \twoheadrightarrow jacm90\,]$

(xi) $?- \;\; mary[jointWorks@phil \twoheadrightarrow Z\,]$

Figure 4: A Sample Database

In the object-oriented systems, a *method* is a function of the form $Objects \times Objects^n \longrightarrow Objects$ or $Objects \times Objects^n \longrightarrow \mathcal{P}(Objects)$, where $\mathcal{P}$ is a power-set operator. Given an object, its methods are “encapsulated” inside that object and constitute its interface to the rest of the system. The first argument of a method is the object the method is being invoked on—the *host object* of the invocation. The other arguments are called *proper* arguments of the invocation.

The reader may have already noticed that double-headed arrows, $\twoheadrightarrow$ and $\Rightarrow\!\!>$, are used in conjunction with set-valued attributes while $\rightarrow$, $\Rightarrow$ signify scalar attributes. Also, double-*shafted* arrows, $\Rightarrow$ and $\Rightarrow\!\!>$, specify types, while arrows $\rightarrow$ and $\twoheadrightarrow$ describe values of attributes. A double-shafted arrow specifies that an attribute (or method) is defined; if it is not given, the single-shafted arrow may not be used. Given $cl[attr \Rightarrow \cdots]$, one cannot use $obj[attr \twoheadrightarrow \cdots]$, where *obj* is in class *cl*, to give a value to *attr* (unless $attr \Rightarrow\!\!> \cdots]$ is specified for some other class where *obj* also belongs). Also, note that (a, b, c)

$empl :: person$ $\quad$ $yuppie :: young$
$student :: person$ $\quad$ $20 : young$
$faculty :: empl$ $\quad$ $30 : yappie$
$child(person) :: person$ $\quad$ $40 : midaged$
$john : student$ $\quad$ $\text{“}CS\text{”} : string$
$john : empl$ $\quad$ $\text{“}Bob\text{”} : string$
$cs_1 : dept$ $\quad$ $alice : child(john)$
$\ddots \quad \ddots$ $\quad$ $\ddots \quad \ddots$

Figure 3: F-logic Representation of the IS-A Hierarchy of Figure 2

when viewed as an object, a class can be a member of another class. For instance, in Figure 2, the classes *string* and *integer* are members of the class *datatype*.

In the actual syntax of F-logic, we use ":" to represent class membership and "::" to denote the subclass relationship. Thus, for instance, the hierarchy of Figure 2 is recorded as shown in Figure 3. Here a statement such as $empl :: person$ says that *empl* is a subclass of *person*; $john : empl$ says that *john* is an instance (*i.e.*, a member) of the class *empl*, and so on. Notice that *john*, *empl*, and *person* are simply terms that denote objects.

The Object Base

Figure 4 presents a database fragment describing employees, students, and others. The first statement there says that the object *bob* is a faculty whose name is "*Bob*". Here *bob* is a logical id of an object that supposedly represents a person. In contrast, "*Bob*" is a member of class *string*; it represents the value of one of *bob*'s attributes, called *name*.

Note that F-logic does not support the dichotomy between "complex objects" and "complex values." We believe that complications introduced by this dichotomy do not justify the benefits (but see Section 15.3). Thus, "*Bob*" is viewed as an oid that represents the string "*Bob*".

Statement (i) also says that *bob* works in the department denoted via the oid cs_1, the department's name is represented by the oid "*CS*", and its manager is described by the object with oid *bob*. Note that *bob* has cyclic references to itself. Statement (ii) represents similar information about *mary*. Unlike the attributes *name*, *highestDegree*, and *age*, which return a single value, the attribute *friends* is *set-valued*. Syntactically, this is indicated by the double-headed arrow "$\twoheadrightarrow$" and the braces "{ }".

Statements (iii) to (vi) provide general information about classes and their *signatures*. A signature of a class specifies names of attributes and methods that are applicable to this class, the type of arguments each method takes, and the type of the result it returns. Statement (iii), for instance, says that for every object in class *faculty*, the attributes *age* and *highestDegree* must be of types *midaged* and *degree*, respectively, and that the attribute *boss* returns results that simultaneously belong to classes *faculty* and *manager*.

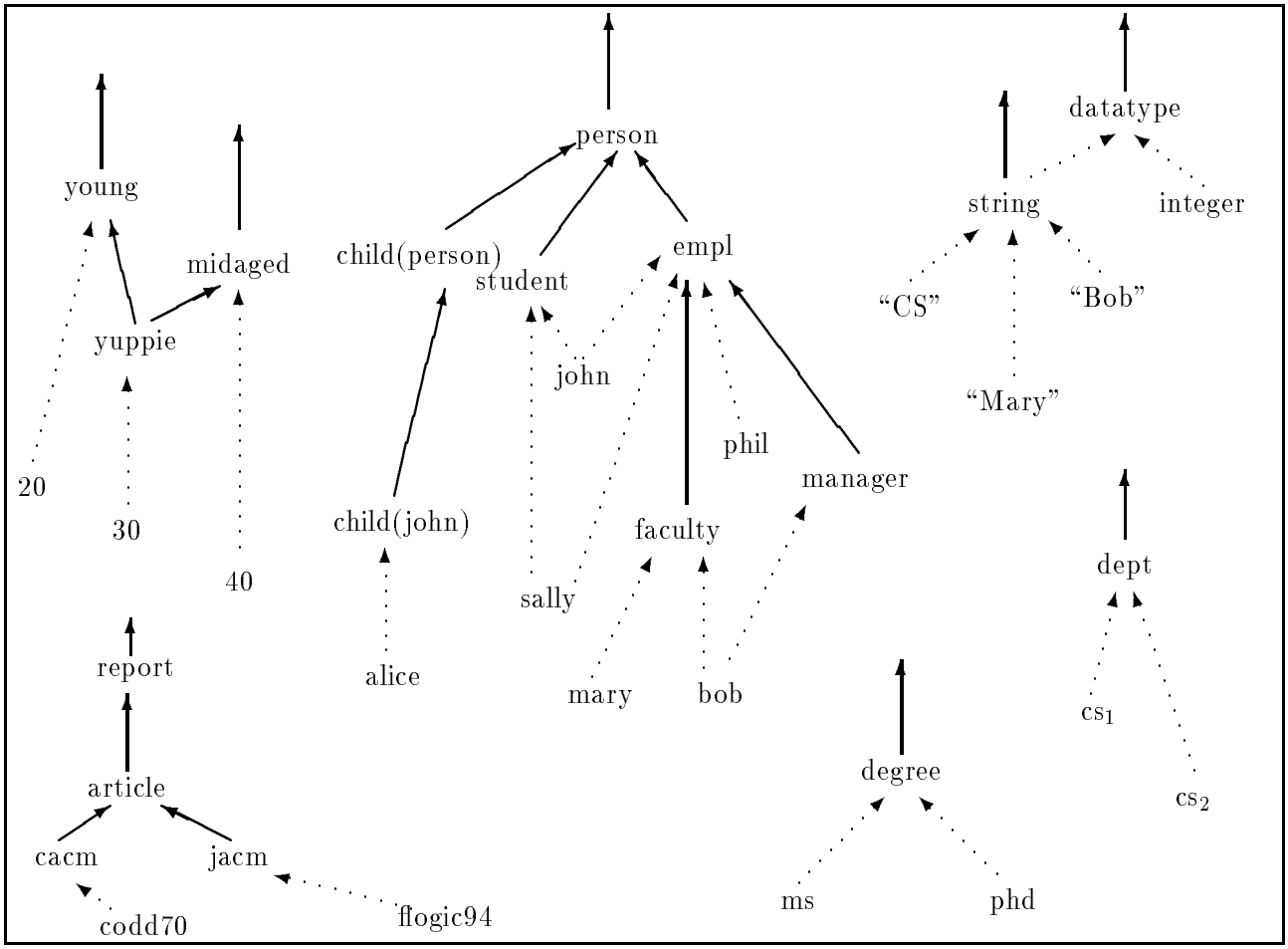

Figure 2: Part of an IS-A Hierarchy

3 F-logic by Example

The development of F-logic is guided by the desire to capture in a logically clean way a number of knowledge representation scenarios. Several of the most salient features of F-logic are described in this section by way of an example.

The IS-A Hierarchy

Figure 2 shows part of a hierarchy of classes and individual objects, where solid arcs represent the subclass relationship and dotted arcs represent class membership. This hierarchy asserts that *faculty* and *manager* are subclasses of *empl*; *student* and *empl* are subclasses of *person*; "*Mary*" and "*CS*" are members of the class *string*; and *mary* is a *faculty*, while *sally* is a member of the class *student*. Note that classes are *reified*, *i.e.*, they belong to the same domain as individual objects. This endows F-logic with a great deal of uniformity, making it possible to manipulate classes and objects in the same language. In particular,

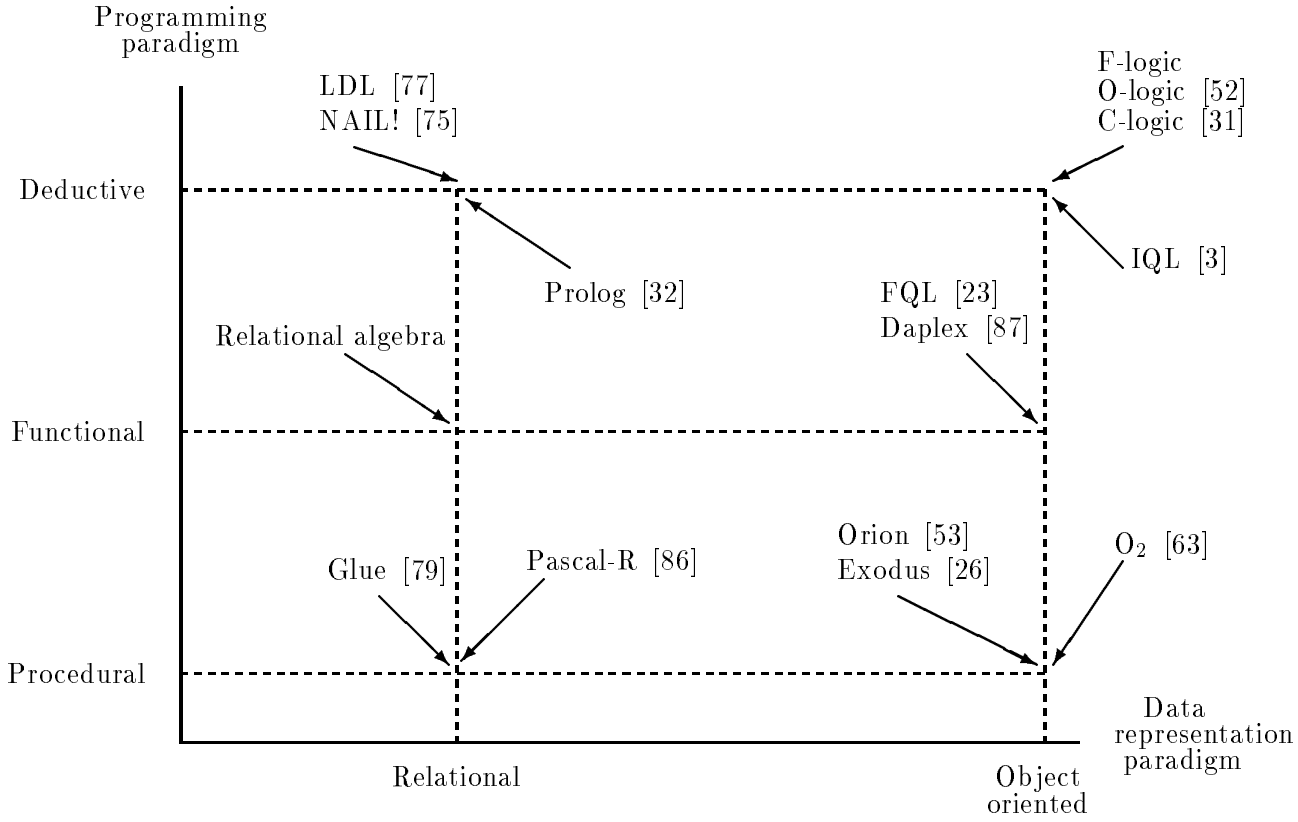

Figure 1: A Classification of Database Programming Languages

We also note that practical considerations sometimes call for a mixture of different paradigms co-existing under the same roof. For instance, Prolog [32] has control primitives that give it a procedural flavor. Logical object-oriented languages [50, 31, 48, 3] allow grouping of data around properties (*i.e.*, relationally) as well as around objects. IQL [3] relies on a small number of procedural features; Pascal-R [86] offers declarative access to data, although the overall structure of the language is procedural.

In general, we believe that the future belongs to multi-paradigm languages, and so the aforementioned "impurity" is not necessarily a drawback. An important research issue is, however, to find ways of *clean* integration of different kinds of languages. It will be clear from this paper that inside the deductive domain, relational and object-oriented languages go hand-in-hand. Interfaces between relational and object-oriented (whether declarative or procedural) languages are also known. However, we are unaware of any clean solution to the integration of procedural object-oriented languages with relational languages; likewise, there is no generally agreed upon framework for absorbing functional languages into the deductive paradigm.

try to put them in a perspective.

It has become customary to define the notion of object-orientation via a long list of properties such systems must have. Some works (*e.g.*, [9]) further divide these properties into those that are "mandatory" and those that are "optional." These classifications, however, do not explain the difference between relational and object-oriented paradigms; nor do they offer justification for the selection of features that they view as important.

We believe that whether a language has an object-oriented flavor or a relational flavor is primarily determined by the way it represents data. In our view, the central feature of the relational data model is that data is conceptually grouped by properties. For instance, information regarding a given person may be scattered among different relations, such as *employee*, *manager*, and *project*, each describing different properties attributed to persons. On the other hand, pieces of data that describe distinct persons via the same property are grouped together in one relation (*e.g.*, $project[empId, projName]$). In contrast, object-oriented representation seeks to group data around objects (again, at the conceptual level). According to this philosophy in language design, the user can access—directly or by inheritance—all public information about any object, once a "handle" to that object is obtained. This handle is usually referred to as *physical object identity*—an implementational notion that has a conceptual, language-level counterpart, which we call *logical object identity*. Logical oid's were introduced in [50] and subsequently utilized in [31, 15] and other works.

The concept of object identity was widely debated in databases. An early attempt to bring this notion into the fold of a logical theory was reported by Maier [65]. Commenting on this work, Ullman [93] concluded that the very concept of object identity is incompatible with logic. We feel that both, Maier's difficulties and Ullman's scepticism, result from a confusion that arose due to lumping physical and logical object identities together into a single concept.

Having decided to group data around objects, it is only natural to try and tap into the potential of such representation by making use of class hierarchies, inheritance, typing, and so on. The difference between "optional" and "mandatory" properties put forward in [9] now becomes a subjective matter of what one perceives to be the most important features of this mode of data representation.

Languages can be classified also along another dimension—their programming paradigm. Here we can point at the following three major groups of languages: procedural, functional, and deductive. Figure 1 shows how various languages may fit into this framework. Our contention is that the misconception about incompatibility of deductive and object-oriented languages comes from overlooking the fact that the two classification axes of Figure 1 are really *orthogonal*. What is commonly referred to as "deductive databases," is simply a class of languages characterized by the flat, relational data model and the deductive programming paradigm. In contrast, most of the systems that are perceived as "object-oriented" are procedural. Only recently, a number of upward-compatible object-oriented logics have been introduced [31, 50, 48], which made it clearer how the perceived incompatibility gap could be bridged. The revised version of F-logic presented here continues this line of research, closing gaps and rectifying flaws of our earlier attempts [50, 48].

retain the spirit of object-oriented programming. In contrast, we propose a logic in which object-oriented concepts are represented directly, both syntactically and semantically.

This work builds on our previous papers [50, 48], which in turn borrowed several important ideas from Maier's O-logic [65] (that, in its turn, was inspired by Aït-Kaci's work on ψ-terms [6, 5]). In [50], we described a logic that adequately covered the structural aspect of complex objects but was short of capturing methods, types, and inheritance. The earlier version of F-logic reported in [48] was a step towards a higher-order syntax. In particular, it supported schema exploration and reasoning about structural inheritance. At the same time, this version of F-logic had several drawbacks as far as its modeling capabilities were concerned.

One of the problems was that all objects were required to form a lattice, which turned out to be impractical for a logic-based language. Another problem was that semantics of attributes was more appropriate for modeling object types rather than their states. All these problems are rectified in the present paper and, in addition, the logic is extended to accommodate types and non-monotonic inheritance.

One aspect of knowledge based systems that is not dealt with here is the issue of updates to database states. Our experience shows that the problem of updates is orthogonal to *structural* aspects of object-oriented systems, and this paper deals with this latter issue only. There has been extensive work on formalizing updates within logic. The reader is referred to [19, 20] for a comprehensive discussion of the problem, for an overview of the related work in the field, and for solutions to many of the previously outstanding problems.

This paper is organized as follows. Section 2 discusses the differences and the similarities between the object-oriented paradigm and the relational paradigm. Section 3 is an informal introduction to some of the main features of the logic. In Sections 4, 5, and 6, we describe the syntax and the semantics of F-logic. Section 7 discusses various semantic properties of the logic. Sections 8, 9, and 10 develop a proof theory for F-logic. Section 11 demonstrates the modeling power of F-logic via a number of non-trivial examples. Section 12 discusses typing. In Section 13 we deal with encapsulation and put forth a novel proposal to view encapsulation as a type-correctness policy. Section 14 presents a semantics for inheritance. An array of issues in data modeling, such as complex values, versions control, and path expressions, is covered in Section 15. Possible extensions to F-logic are discussed in Section 16. In Section 17 we provide a retrospective view of the internal structure of F-logic and relate it to classical predicate calculus. Section 18 concludes the paper.

2 A Perspective on Object-Oriented vs. Declarative Programming

A number of researchers had argued that object-oriented languages are fundamentally different and even incompatible with other paradigms, especially with logic programming [93, 67]. The ensuing debate was a reflection of the lack of early success in formalizing many aspects of object-oriented programming. Another reason was that there was no framework in which to classify various approaches, so that their differences and common points could be seen in a perspective. In this section we address these issues and

1 Introduction

The past decade was marked by a considerable interest in the object-oriented approach, both within the database community and among researchers in programming languages. Although the very term "the object-oriented approach" is defined fairly loosely, a number of concepts, such as complex objects, object identity, methods, encapsulation, typing and inheritance, have been identified as the most salient features of that approach [12, 88, 103, 97, 9].

One of the important driving forces behind the interest in object-oriented languages in databases is the promise they show in overcoming the, so called, *impedance mismatch* [66, 103] between programming languages for writing applications and languages for data retrieval. Concurrently, a different, *deductive* approach has gained enormous popularity. Since logic can be used as a computational formalism *and* as a data specification language, proponents of the deductive approach have argued that this approach, too, overcomes the aforesaid mismatch problem. However, in their present form, both approaches have shortcomings. One of the main problems with the object-oriented approach is the lack of logical semantics, which traditionally was important for database programming languages. On the other hand, deductive databases normally use a flat data model and do not support data abstraction. It therefore can be expected that combining the two paradigms will pay off in a big way.

A great number of attempts to combine the two approaches has been reported in the literature (see, *e.g.*, [1, 2, 3, 13, 16, 17, 31, 50, 60, 58, 65, 85, 10]) but, in our opinion, none was entirely successfully. These approaches would seriously restrict object structure and queries; or they may sacrifice declarativeness by adding extra-logical concepts; or they would not address certain important aspects of object-oriented systems, such as typing or inheritance.

In this paper we propose a formalism, called *Frame Logic* (abbr., F-logic), that achieves *all* of the goals listed above and, in addition, is suitable for defining and manipulating database schema and types. F-logic is a *full-fledged* logic; it has a model-theoretic semantics and a sound and complete proof theory. In a sense, F-logic stands in the same relationship to the object-oriented paradigm as classical predicate calculus stands to relational programming.

Besides object-oriented databases, another important application for F-logic is in the area of frame-based languages in AI [37, 72], since these languages are also built around the concepts of complex objects, inheritance, and deduction. It is from this connection that the name "Frame Logic" was derived. However, most of our terminology comes from the object-oriented parlance, not from AI. Thus, we will be talking about objects and attributes instead of frames, slots, and the like.

To reason about inheritance and for tasks requiring exploration of the knowledge base schema, a logic-based language would be greatly aided by higher-order features. However, higher-order logics must be approached with great care to ensure the desired computational properties. In the past, a number of researchers suggested that many useful higher-order concepts of knowledge representation languages can be encoded in predicate calculus [41, 69]. From a programmer's point of view, however, encoding does not adequately capture many higher-order constructs, as it gives no *direct* semantics and does not

Contents

Logical Foundations of Object-Oriented and Frame-Based Languages

Michael Kifer * Georg Lausen † James Wu ‡

Abstract

We propose a novel logic, called Frame Logic (abbr., F-logic), that accounts in a clean, declarative fashion for most of the structural aspects of object-oriented and frame-based languages. These features include object identity, complex objects, inheritance, polymorphic types, methods, encapsulation, and others. In a sense, F-logic stands in the same relationship to the object-oriented paradigm as classical predicate calculus stands to relational programming. The syntax of F-logic is higher-order, which, among other things, allows the user to explore data and schema using the same declarative language. F-logic has a model-theoretic semantics and a sound and complete resolution-based proof procedure. This paper also discusses various aspects of programming in declarative object-oriented languages based on F-logic.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Languages—*query languages*; I.2.3 [**Artificial Intelligence**]: Deduction and theorem proving—*deduction, logic programming, nonmonotonic reasoning*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical logic—*logic programming, mechanical theorem proving*

General Terms: Languages, Theory

Additional Key Words and Phrases: Object-oriented programming, frame-based languages, deductive databases, logic programming, semantics, proof theory, typing, nonmonotonic inheritance

Technical Report 93/06[1]
April 1993

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794

*Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, U.S.A.
Email: kifer@cs.sunysb.edu. Work supported in part by the NSF grants DCR-8603676 and IRI-8903507.

†Fakultät für Mathematik und Informatik, Universität Mannheim, D-68131 Mannheim, Germany.
Email: lausen@pi3.informatik.uni-mannheim.de.

‡Renaissance Software, 175 S. San Antonio Rd., Los Altos, CA 94022, U.S.A. Email: wu@rs.com.
Work supported in part by the NSF grant IRI-8903507.

[1] A substantial revision of Technical Report 90/14 bearing the same title.