[22] T. Sheard and L. Fegaras. A Fold for All Seasons. *Sixth Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pp 233–242, June 1993.

[23] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Proceedings of the 2nd European Symposium on Programming, Nancy, France*, pp 344–358, March 1988. LNCS 760.

[24] J. B. Wells. Typability and Type Checking in the Second-Order λ-Calculus Are Equivalent and Undecidable. *Proceedings of the 1994 IEEE Symposium on Logic in Computer Science*, pp 176–185, 1994.

[4] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[5] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[6] W. Chin. Safe Fusion of Functional Expressions. *Proceedings of the ACM Symposium on Lisp and Functional Programming, San Francisco, California*, pp 11–20, June 1992.

[7] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[8] R. Cockett and T. Fukushima. About Charity. Technical report, Department of Computer Science, the University of Calgary, Alberta, Canada, June 1992. Research Report No. 92/480/18.

[9] L. Fegaras. Efficient Optimization of Iterative Queries. In *Fourth International Workshop on Database Programming Languages, Manhattan, New York City*, pp 200–225. Springer-Verlag, Workshops on Computing, August 1993.

[10] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. *ACM SIGMOD International Conference on Management of Data, San Jose, California*, pp 47–58, May 1995.

[11] P. Freyd and A. Scedrov. Some Semantic Aspects of Polymorphic Lambda Calculus. *Proceedings of the 1987 IEEE Symposium on Logic in Computer Science*, pp 315–319, 1987.

[12] A. Gill, J. Launchbury, and S. Peyton Jones. A Short Cut to Deforestation. *Sixth Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pp 223–232, June 1993.

[13] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[14] G. Hillebrand and P. Kanellakis. Functional Database Query Languages as Typed Lambda Calculi of Fixed Order. *Proceedings of the 13th ACM Symposium on Principles of Database Systems, Minneapolis, MN*, pp 222–231, May 1994.

[15] G. Hillebrand, P. Kanellakis, and H. Mairson. Database Query Languages Embedded in the Typed Lambda Calculus. *Proceedings of the 1993 IEEE Symposium on Logic in Computer Science*, pp 332–343, 1993.

[16] R. Kieburtz and L. Jeffrey. Programming with Algebras. *Advanced Functional Programming*, pp 267–307, 1995.

[17] P. Lee, M. Leone, S. Michaylov, and F. Pfenning. Towards a Practical Programming Language Based on the Polymorphic Lambda Calculus. Technical report, School of Computer Science, Carnegie Mellon University, November 1989. Ergo Project Report ERGO-89-085.

[18] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts*, pp 124–144, August 1991. LNCS 523.

[19] L. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[20] B. Pierce, S. Dietzen, and S. Michaylov. Programming in Higher-Order Typed Lambda-Calculi. Technical report, Carnegie Mellon University, March 1989. CMU-CS-89-111. Available at `http://www.cl.cam.ac.uk/users/bcp1000/ftp/leap.ps.gz`.

[21] J. C. Reynolds. An Introduction to the Polymorphic Lambda Calculus. *Logical Foundations of Functional Programming*, 1988.

4 Conclusion and Future Work

There has been much recent interest in capturing multiple collection types in a single database language. Earlier approaches proposed a separate, self-contained, calculus for each collection type along with a number of coercion operators to transform one collection type to another [4]. These approaches require that a list be converted to a set before it is joined with a set. In our earlier work [10], we expressed a fixed number of collection types as monoids. Even though this approach allows us to mix operations over multiple collection types in a safe way, it does not support user-defined collection types.

In this paper we followed a novel approach: we created two new type constructors that capture the commutativity and idempotence properties of many user-defined collection types. We presented a small set of typing rules, which can be used to filter out all inconsistent programs and we proved the soundness of the typing rules.

In conclusion, we have presented a database calculus, F_{db}, based on F_2 that

- captures multiple collection types, such as sets, bags, lists, and vectors, which can be arbitrary nested;
- is type sound and the type-checking algorithm aborts all inconsistent programs, such as converting a set into a list;
- captures all primitive recursive functions;
- supports a normalization algorithm that consists of a small number of reduction rules and generalizes many earlier algebraic optimization algorithms for queries.

A common criticism against F_2 is that it requires a large amount of type annotations, which make it impractical as a real programming language. It has been proved that type reconstruction (i.e., inferring the types of F_2 terms when some type annotations are missing) is undecidable in general [24]. But there are languages that partially solve this problem. For example, SML does not require type annotations at all. In fact it does not allow explicit universal quantification over types. The parametric polymorphism inherent in SML is achieved by using a generic 'let' construct (called let-polymorphism). For future work, we intend to explore this idea of defining a database language without explicit type annotations that captures the same database collection types as F_{db}.

References

[1] S. Bellantoni and S. Cook. A new Recursion-Theoretic Characterization of the Polynomial Time. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia*, pp 283–293, May 1992.

[2] C. Bohm and A. Berarducci. Automatic Synthesis of Typed Λ-Programs on Term Algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[3] V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. In *18th International Colloquium on Automata, Languages and Programming, Madrid, Spain*, pp 60–75. Springer-Verlag, July 1991. LNCS 510.

The following are examples of common operations over bags that people have proposed. The boxed operations are the problematic expressions that are filtered out by our typing rules.

$$\begin{array}{lcl}
\text{bag-map} & = & \Lambda\beta.\ \Lambda\delta.\ \lambda f:\beta\rightarrow\delta.\ \lambda x:\text{bag}(\beta).\ \Lambda\alpha.\ \lambda[c:\delta\Rightarrow\alpha, n:\alpha].\\
 & & x[\alpha]\ (\lambda[y:\beta, r:\alpha].\ c\ (f\ y)\ r)\ n\\
\text{bag-cardin} & = & \Lambda\beta.\ \lambda x:\text{bag}(\beta).\ \Lambda\alpha.\ \lambda[s:()\Rightarrow\alpha, z:\alpha].\ x[\alpha]\ (\lambda[y:\beta, r:\alpha].\ s\ ()\ r)\ z\\
\text{list>bag} & = & \Lambda\beta.\ \lambda x:\text{list}(\beta).\ \Lambda\alpha.\ \lambda[c:\beta\Rightarrow\alpha, n:\alpha].\ x[\alpha]\ (\lambda y:\beta.\ \lambda r:\alpha.\ c\ y\ r)\ n\\
\text{bag>list} & = & \Lambda\beta.\ \lambda x:\text{bag}(\beta).\ \Lambda\alpha.\ \lambda c:\beta\rightarrow\alpha\rightarrow\alpha.\ \lambda n:\alpha.\ x[\alpha]\ (\lambda[y:\beta, r:\alpha].\ \boxed{c\ y\ r}\ )\ n\\
\text{choose} & = & \Lambda\beta.\ \lambda x:\text{bag}(\beta).\ \lambda d:\beta.\ x[\beta]\ (\lambda[y:\beta, r:\beta].\ \boxed{y}\ )\ d
\end{array}$$

where $\boxed{c\ y\ r}$ indicates a type error because $c\ y$ of type $\alpha\rightarrow\alpha$ is applied to r of type $\oplus\alpha$, which is not acceptable by our typing rules (according to Rules T5 and T6, only $\oplus\alpha\rightarrow\oplus\alpha$ or $\oplus\alpha\rightarrow\otimes\alpha$ can be applied to $\oplus\alpha$). The $\boxed{y}$ is another type error because Rule T1 expects y to be either of type $\oplus\beta$ or $\otimes\beta$. These two operations, bag>list (that converts a bag into a list) and choose (that selects a bag element), are the ones we want to rule out, because they are nondeterministic.

Set operations in F_{db} can capture relational and nested relational operators:

$$\begin{array}{lcl}
\text{join} & = & \Lambda\beta.\ \Lambda\delta.\ \Lambda\gamma.\ \lambda p:\beta\rightarrow\delta\rightarrow\text{bool}.\ \lambda f:\beta\rightarrow\delta\rightarrow\gamma.\ \lambda x:\text{set}(\beta).\ \lambda y:\text{set}(\delta).\\
 & & \Lambda\alpha.\ \lambda[c:[\gamma,()]|\!\!\Rightarrow\alpha, u:(), n:\alpha].\ x[\alpha]\ (\lambda[z:\beta, u:(), r:\alpha].\ y[\alpha]\ (\lambda[w:\delta, u:(), s:\alpha].\\
 & & \text{if}[\alpha]\ (p\ z\ w)\ (c\ (\text{pair}(f\ z\ w, ()))\ s)\ s)\ r)\ n\\
\text{flatten} & = & \Lambda\beta.\ \lambda x:\text{set}(\text{set}(\beta)).\ \Lambda\alpha.\ \lambda[c:[\beta,()]|\!\!\Rightarrow\alpha, u:(), n:\alpha].\ x[\alpha]\ (\lambda[y:\text{set}(\beta), u:(), r:\alpha].\ y[\alpha]\ c\ r)\ n\\
\text{cardin} & = & \Lambda\beta.\ \lambda x:\text{set}(\beta).\ \Lambda\alpha.\ \lambda[s:()\Rightarrow\alpha, z:\alpha].\ x[\alpha]\ (\lambda[y:\beta, u:(), r:\alpha].\ \boxed{s\ ()\ r}\ )\ z\\
\text{bag>set} & = & \Lambda\beta.\ \lambda x:\text{bag}(\beta).\ \Lambda\alpha.\ \lambda[c:[\beta,()]|\!\!\Rightarrow\alpha, u:(), n:\alpha].\ x[\alpha]\ (\lambda[y:\beta, r:\alpha].\ c\ (\text{pair}(y,()))\ r)\ n\\
\text{set>bag} & = & \Lambda\beta.\ \lambda x:\text{set}(\beta).\ \Lambda\alpha.\ \lambda[c:\beta\Rightarrow\alpha, n:\alpha].\ x[\alpha]\ (\lambda[y:\beta, u:(), r:\alpha].\ \boxed{c\ y\ r}\ )\ n\\
\text{forall} & = & \Lambda\beta.\ \lambda p:\beta\rightarrow\text{bool}.\ \lambda x:\text{set}(\beta).\ x[\text{bool}]\ (\lambda[y:\beta, u:(), r:\text{bool}].\ r[\text{bool}]\ (p\ y)\ \text{false})\ \text{true}
\end{array}$$

The $\boxed{s\ ()\ r}$ and $\boxed{c\ y\ r}$ indicate type errors because $(s\ ())$ and $c\ y$ of type $\oplus\alpha\rightarrow\oplus\alpha$ are applied to r of type $\otimes\alpha$ (according to Rule T6, only $\otimes\alpha\rightarrow\otimes\alpha$ can be applied to $\otimes\alpha$).

Theorem 2 *The F_{db} typing rules are sound.*

Proof sketch: We have the following axioms (for all t, t_1, t_2, t_3):

$$\mathcal{D}[\![\otimes t]\!] \subseteq \mathcal{D}[\![\oplus t]\!] \subseteq \mathcal{D}[\![t]\!] \tag{5}$$

$$\mathcal{D}[\![t_1 \Rightarrow t_2]\!] \subseteq \mathcal{D}[\![t_1 \rightarrow \oplus t_2 \rightarrow \oplus t_2]\!] \tag{6}$$

$$\mathcal{D}[\![[t_1, t_2]|\!\!\Rightarrow t_3]\!] \subseteq \mathcal{D}[\![\text{pair}(t_1, t_2) \rightarrow \otimes t_3 \rightarrow \otimes t_3]\!] \tag{7}$$

To prove the soundness of the typing rules, we use the same technique as in the proof of Th. 1. For example, the T5 rule is mapped into the predicate:

$$\mathcal{T}[\![\Gamma]\!] \ \Rightarrow\ (e_1 \in \mathcal{D}[\![\oplus t \rightarrow \oplus t]\!] \ \wedge\ e_2 \in \mathcal{D}[\![\diamond t]\!] \ \Rightarrow\ (e_1\ e_2) \in \mathcal{D}[\![\oplus t]\!])$$

for $\diamond \in \{\oplus, \emptyset\}$. This is a tautology because:

$$\begin{array}{ll}
e_1 \in \mathcal{D}[\![\oplus t \rightarrow \oplus t]\!] \ \wedge\ e_2 \in \mathcal{D}[\![\diamond t]\!] & \\
\equiv\ (\forall x:\ x \in \mathcal{D}[\![\oplus t]\!] \ \Rightarrow\ (e_1\ x) \in \mathcal{D}[\![\oplus t]\!]) \ \wedge\ e_2 \in \mathcal{D}[\![\diamond t]\!] & \text{from Eq. 1}\\
\Rightarrow\ (e_1\ e_2) \in \mathcal{D}[\![\oplus t]\!] & \text{from Eq. 5}
\end{array}$$

□

$$
\begin{array}{lcl}
t, t_1, t_2, t_3 & ::= & v \;\mid\; \forall v.\, t \;\mid\; t_1 \rightarrow t_2 \;\mid\; t_1 \Rightarrow t_2 \;\mid\; [t_1, t_2] \Mapsto t_3 \\
e, e_1, e_2 & ::= & x \;\mid\; e_1\, e_2 \;\mid\; \lambda x : t.\, e \;\mid\; \Lambda v.\, e \;\mid\; e[t] \;\mid\; \lambda[x_1 : t_1, x_2 : t_2].\, e \;\mid\; \lambda[x_1 : t_1, x_2 : t_2, x_3 : t_3].\, e \\
\Gamma & ::= & <> \;\mid\; x : t, \Gamma \;\mid\; x : \oplus t, \Gamma \;\mid\; x : \otimes t, \Gamma \\
\Box & ::= & \oplus \;\mid\; \otimes
\end{array}
$$

$$
\text{(T1)}\quad \frac{\Gamma, x_1 : t_1, x_2 : \oplus t_2 \vdash e : \oplus t_2}{\Gamma \vdash (\lambda[x_1 : t_1, x_2 : t_2].\, e) : t_1 \Rightarrow t_2}
\qquad
\text{(T5)}\quad \frac{\Gamma \vdash e_1 : \oplus t \rightarrow \oplus t \qquad \Gamma \vdash e_2 : \diamond t \qquad \diamond \in \{\oplus, \emptyset\}}{\Gamma \vdash (e_1\, e_2) : \oplus t}
$$

$$
\text{(T2)}\quad \frac{\Gamma, x_1 : t_1, x_2 : t_2, x_3 : \otimes t_3 \vdash e : \otimes t_3}{\Gamma \vdash (\lambda[x_1 : t_1, x_2 : t_2, x_3 : t_3].\, e) : [t_1, t_2] \Mapsto t_3}
\qquad
\text{(T6)}\quad \frac{\Gamma \vdash e_1 : \otimes t \rightarrow \otimes t \qquad \Gamma \vdash e_2 : \diamond t \qquad \diamond \in \{\oplus, \otimes, \emptyset\}}{\Gamma \vdash (e_1\, e_2) : \otimes t}
$$

$$
\text{(T3)}\quad \frac{\Gamma \vdash e_1 : t_1 \Rightarrow t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\, e_2) : \oplus t_2 \rightarrow \oplus t_2}
\qquad
\text{(T7)}\quad \frac{\Gamma \vdash e_1 : \Box(t_1 \rightarrow t_2) \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\, e_2) : \Box t_2}
$$

$$
\text{(T4)}\quad \frac{\Gamma \vdash e_1 : [t_1, t_2] \Mapsto t_3 \qquad \Gamma \vdash e_2 : \mathrm{prod}(t_1, t_2)}{\Gamma \vdash (e_1\, e_2) : \otimes t_3 \rightarrow \otimes t_3}
\qquad
\text{(T8)}\quad \frac{\Gamma \vdash e : \Box(\forall v.\, t_1)}{\Gamma \vdash (e[t_2]) : \Box([t_2/v]t_1)}
$$

Figure 2: Types and terms in $\mathrm{F_{db}}$ and the typing rules

If $s : \mathrm{set}(\beta)$, then $(x = y \;\Rightarrow\; c\,x\,(c\,y\,z) = c\,x\,z) \;\Rightarrow\; s\,c\,(s\,c\,n) = s\,c\,n$. That is, the set key is the entire set element of type β. A more interesting example is a set of employees with the employee's social security number as the primary key:

$$
\mathrm{employees} \;=\; \forall\alpha.\, [\,[\mathrm{int}, \mathrm{pair}(\mathrm{int}, \mathrm{int})] \Mapsto \alpha, ()] \Mapsto \alpha
$$

where pair(int, int) contains the non-key attributes of the employee.

3.3 The Typing Rules

Figure 2 presents the meta-syntax of the types and the terms in $\mathrm{F_{db}}$ along with the typing rules that extend the typing rules in Figure 1. The new λ-abstractions $\lambda[x_1 : t_1, x_2 : t_2].\, e$ and $\lambda[x_1 : t_1, x_2 : t_2, x_3 : t_3].\, e$ create functions of types $t_1 \Rightarrow t_2$ and $[t_1, t_2] \Mapsto t_3$, respectively. The operational meaning of these λ-abstractions is straightforward:

$$
\begin{array}{lcl}
\lambda[x_1 : t_1, x_2 : t_2].\, e & \Rightarrow & \lambda x_1 : t_1.\, \lambda x_2 : t_2.\, e \\
\lambda[x_1 : t_1, x_2 : t_2, x_3 : t_3].\, e & \Rightarrow & \lambda x : \mathrm{prod}(t_1, t_2).\, \lambda x_3 : t_3.\, [(\mathrm{fst}\, x)/x_1, (\mathrm{snd}\, x)/x_2]e
\end{array}
$$

where x is a variable that does not appear free in e.

The types in the type binding Γ, as well as the types of the type assertions, may be annotated by $\oplus$ or $\otimes$. Rules T5 and T6 indicate that if a commutative (resp. commutative and idempotent) function is expected, then either a commutative or a plain function (resp. any function) can be passed. Rules T1 and T2 create the annotations from the new type constructors. The other rules simply propagate the annotations.

This equation indicates that any two values f, g of $\mathcal{D}[\![t_1 \Rightarrow t_2]\!]$ satisfy $f\,x\,(g\,y\,z) = g\,y\,(f\,x\,z)$, for all x, y, z. This implies that every value, f, of $\mathcal{D}[\![t_1 \Rightarrow t_2]\!]$ is a commutative function: i.e., $f\,x\,(f\,y\,z) = f\,y\,(f\,x\,z)$, for all x, y, z. Using this type constructor, the bag type can be represented as follows:

$$\mathrm{bag}(\beta) \;=\; \forall\alpha.\,(\beta \Rightarrow \alpha) \Rightarrow \alpha$$

That is, if we instantiate α to t, we get:

$$\begin{aligned}\mathcal{D}[\![(\beta \Rightarrow t) \Rightarrow t]\!] &= \{\, f, g \mid\; \forall c, c' \in \mathcal{D}[\![\beta \Rightarrow t]\!] :\; \forall n \in \mathcal{D}[\![t]\!] :\; f\,c\,(g\,c'\,n) = g\,c'\,(f\,c\,n) \,\}\\ &= \{\, f, g \mid\; \forall c, c' :\; (\forall x, y \in \mathcal{D}[\![\beta]\!] :\; \forall z \in \mathcal{D}[\![t]\!] :\; c\,x\,(c'\,y\,z) = c'\,y\,(c\,x\,z))\\ &\qquad\qquad \Rightarrow\; \forall n \in \mathcal{D}[\![t]\!] :\; f\,c\,(g\,c'\,n) = g\,c'\,(f\,c\,n) \,\}\end{aligned}$$

This implies that if a function c (which corresponds to the bag insert operation) is commutative (i.e., $c\,x\,(c\,y\,z) = c\,y\,(c\,x\,z)$), then the bag iteration $(x\,c\,n)$ satisfies $x\,c\,(y\,c\,n) = y\,c\,(x\,c\,n)$. Instances of the type $t_1 \Rightarrow t_2$ can be constructed by the special λ-abstraction $\lambda[x_1 : t_1, x_2 : t_2].\, e$. For instance, the bag $x = \{\!\{1, 2\}\!\}$ is represented by $\Lambda\alpha.\,\lambda[c : \mathrm{int} \Rightarrow \alpha, n : \alpha].\, c\,1\,(c\,2\,n)$ and the bag $y = \{\!\{3, 4\}\!\}$ by $\Lambda\alpha.\,\lambda[c : \mathrm{int} \Rightarrow \alpha, n : \alpha].\, c\,3\,(c\,4\,n)$. If c is commutative, then, according to the previous definition, $x[\alpha]\;c\;(y[\alpha]\;c\;n) = y[\alpha]\;c\;(x[\alpha]\;c\;n)$. That is, $c\,1\,(c\,2\,c\,3\,(c\,4\,n)) = c\,3\,(c\,4\,c\,1\,(c\,2\,n))$. Consequently, the bag union defined by

$$x \uplus y \;=\; \Lambda\alpha.\,\lambda[c : \beta \Rightarrow \alpha, n : \alpha].\, x[\alpha]\;c\;(y[\alpha]\;c\;n)$$

for $x, y : \mathrm{bag}(\beta)$, is commutative, i.e., $x \uplus y = y \uplus x$.

The integer type can be coded as bag(()):

$$\mathrm{int} \;=\; \forall\alpha.\,(() \Rightarrow \alpha) \Rightarrow \alpha$$

(Recall that () is the unit type $\forall\alpha : \alpha \rightarrow \alpha$, which has only one instance: the identity function.) This coding of integers is isomorphic to the type $\forall\alpha.\,(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$.

3.2 Idempotence

We introduce another type constructor $[\,t_1, t_2]\!\!\Rightarrow t_3$ that captures types that are both commutative and idempotent. Its domain satisfies $\mathcal{D}[\![\,[\,t_1, t_2]\!\!\Rightarrow t_3]\!] \subseteq \mathcal{D}[\![\mathrm{prod}(t_1, t_2) \Rightarrow t_3]\!] \subseteq \mathcal{D}[\![\mathrm{prod}(t_1, t_2) \rightarrow t_3 \rightarrow t_3]\!]$. More specifically, we have:

$$\begin{aligned}\mathcal{D}[\![\,[\,t_1, t_2]\!\!\Rightarrow t_3]\!] &= \{\, f \mid f \in \mathcal{D}[\![\mathrm{prod}(t_1, t_2) \Rightarrow t_3]\!] \;\wedge\; \forall x, y \in \mathcal{D}[\![\mathrm{prod}(t_1, t_2)]\!] : \forall z \in \mathcal{D}[\![t_3]\!] :\\ &\qquad (\mathrm{fst}\,x) = (\mathrm{fst}\,y) \;\Rightarrow\; f\,x\,(f\,y\,z) = f\,x\,z \,\}\end{aligned} \tag{4}$$

where $\mathrm{prod}(t_1, t_2)$ is the product of the types t_1 and t_2: the first component of type t_1 is the set key while the second is a non-key element. If two elements x and y have the same key (i.e., $(\mathrm{fst}\,x) = (\mathrm{fst}\,y)$), then they satisfy $f\,x\,(f\,y\,z) = f\,x\,z$. That is, if we insert two values with the same key in a set, only the last value is kept in.

For example, sets can be captured as follows:

$$\mathrm{set}(\beta) \;=\; \forall\alpha.\,[\,[\,\beta, ()]\!\!\Rightarrow \alpha, ()]\!\!\Rightarrow \alpha$$

The F_2 reduction rules can be used for program fusion [6] and deforestation (the elimination of intermediate data structures) [23]. That is, when one program produces an intermediate data structure as output and another program consumes this data structure as input, we can fuse these two program in such a way that the intermediate data structure is no longer produced. Consider for example the fusion of map with filter: (where $f : \beta \rightarrow \gamma$, $p : \gamma \rightarrow \text{bool}$, and $x : \text{list}(\beta)$)

$$\begin{array}{ll}
\multicolumn{2}{l}{\text{filter}[\gamma](p)(\text{map}[\beta][\gamma](f)\,x)} \\
= & \Lambda\alpha.\,\lambda c : \beta\rightarrow\alpha\rightarrow\alpha.\,\lambda n : \alpha.\,(\Lambda\alpha.\,\lambda c : \gamma\rightarrow\alpha\rightarrow\alpha.\,\lambda n : \alpha.\,x[\alpha]\,(\lambda z : \beta.\,\lambda r : \alpha.\,c\,(f\,z)\,r)\,n)[\alpha] \\
 & (\lambda z : \beta.\,\lambda r : \alpha.\,(p\,z)[\alpha]\,(c\,z\,r)\,r)\,n \\
\Rightarrow^t_\beta & \Lambda\alpha.\,\lambda c : \beta\rightarrow\alpha\rightarrow\alpha.\,\lambda n : \alpha.\,(\lambda c : \gamma\rightarrow\alpha\rightarrow\alpha.\,\lambda n : \alpha.\,x[\alpha]\,(\lambda z : \beta.\,\lambda r : \alpha.\,c\,(f\,z)\,r)\,n) \\
 & (\lambda z : \beta.\,\lambda r : \alpha.\,(p\,z)[\alpha]\,(c\,z\,r)\,r)\,n \\
\Rightarrow_\beta & \Lambda\alpha.\,\lambda c : \beta\rightarrow\alpha\rightarrow\alpha.\,\lambda n : \alpha.\,x[\alpha]\,(\lambda z : \beta.\,\lambda r : \alpha.\,(\lambda z : \beta.\,\lambda r : \alpha.\,(p\,z)[\alpha]\,(c\,z\,r)\,r)\,(f\,z)\,r)\,n \\
\Rightarrow_\beta & \Lambda\alpha.\,\lambda c : \beta\rightarrow\alpha\rightarrow\alpha.\,\lambda n : \alpha.\,x[\alpha]\,(\lambda z : \beta.\,\lambda r : \alpha.\,(p\,(f\,z))[\alpha]\,(c\,(f\,z)\,r)\,r)\,n
\end{array}$$

Notice that the list created by map in the original program is no longer produced in the normalized program. The β- and η-reduction rules can also be used for proving program equivalences without the need of inductive proofs: two functions are extensionally equal if their normal forms are α-equivalent (i.e., they are equal under variable renaming).

3 $\mathbf{F}_{\text{db}}$: The Polymorphic Database Calculus

Bags and sets can be considered as lists with some additional properties [3, 10]. For example, the list append function is associative but the bag union is both associative and commutative, and the set union is associative, commutative, and idempotent (i.e., the union of a set with itself is the same set). These additional properties make the list representation inappropriate for bags and sets, because of possible inconsistencies that are introduced by some inherently nondeterministic operations. For example, there are infinite ways of converting a set into a bag (e.g., the set $\{1\}$ can be converted into the bags $\{\!\{1\}\!\}$, $\{\!\{1,1\}\!\}$, $\{\!\{1,1,1\}\!\}$, etc.), and many ways of converting a bag to a list (if all the possible orders are considered). Since F_2 is purely deterministic, we would like to avoid such cases. In addition, by representing bags and sets as lists, the additional properties of these types cannot be used for reasoning and optimization. There some optimizations for bags that are not possible for lists, such as rearranging join orders.

In this section we extend the type system of F_2 to capture types with commutativity and idempotence properties. We present a set of typing rules for these types, which accept legitimate types only, and we prove the soundness of these rules.

3.1 Commutativity

We introduce a new type constructor $t_1 \Rightarrow t_2$ that captures commutative types. Its domain satisfies $\mathcal{D}[\![t_1 \Rightarrow t_2]\!] \subseteq \mathcal{D}[\![t_1 \rightarrow t_2 \rightarrow t_2]\!]$. More specifically, we have:

$$\begin{array}{rcl} \mathcal{D}[\![t_1 \Rightarrow t_2]\!] & = & \{\; f,\,g \mid f, g \in \mathcal{D}[\![t_1 \rightarrow t_2 \rightarrow t_2]\!] \;\wedge\; \forall x, y \in \mathcal{D}[\![t_1]\!] : \forall z \in \mathcal{D}[\![t_2]\!] : \\ & & \qquad f\,x\,(g\,y\,z) = g\,y\,(f\,x\,z)\;\} \end{array} \qquad (3)$$

Church numeral is also an iteration scheme to manipulate this numeral. For example, (4 plus 3) calculates $4+3$ since it is equivalent to succ(succ(succ(succ 3))). This iteration scheme (also called *fold* [9, 22] and *catamorphism* [18]) is the only way to manipulate integers in F_2.

Church numerals can capture all primitive recursive functions, including predecessor, difference, and integer equality. But Church numerals go beyond primitive recursion when computing higher-order values, that is, when doing second order arithmetic (for more results on expressiveness, the reader is referred to earlier work [13, 14, 15, 1]). For example, the Ackermann's function is coded as an iterator that computes values of type int$\rightarrow$int:

$$\text{ack} \;=\; \lambda m:\text{int}.\ \lambda n:\text{int}.\ x[\text{int}\rightarrow\text{int}]\ (\lambda f:\text{int}\rightarrow\text{int}.\ \lambda k:\text{int}.\ (\text{succ}\ k)[\text{int}]\ (\text{succ zero})\ f)\ \text{succ}\ n$$

Lists can be defined in a way similar to Church numerals:

$$\text{list}(\beta) \;=\; \forall\alpha.\ (\beta\rightarrow\alpha\rightarrow\alpha)\rightarrow\alpha\rightarrow\alpha$$

The first component of the list type, $(\beta\rightarrow\alpha\rightarrow\alpha)$, corresponds to the cons constructor while the second α to the nil value. For example, the list $[1,2,3]$ is given by

$$\Lambda\alpha.\ \lambda c:\text{int}\rightarrow\alpha\rightarrow\alpha.\ \lambda n:\alpha.\ c\ 1\ (c\ 2\ (c\ 3\ n))$$

The following are examples of list operations:

$$\begin{array}{lcl}
\text{nil} & = & \Lambda\beta.\ \Lambda\alpha.\ \lambda c:\beta\rightarrow\alpha\rightarrow\alpha.\ \lambda n:\alpha.\ n \\
\text{cons} & = & \Lambda\beta.\ \lambda x:\beta.\ \lambda r:\text{list}(\beta).\ \Lambda\alpha.\ \lambda c:\beta\rightarrow\alpha\rightarrow\alpha.\ \lambda n:\alpha.\ c\ x\ (r[\alpha]\ c\ n) \\
\text{append} & = & \Lambda\beta.\ \lambda x:\text{list}(\beta).\ \lambda y:\text{list}(\beta).\ \Lambda\alpha.\ \lambda c:\beta\rightarrow\alpha\rightarrow\alpha.\ \lambda n:\alpha.\ x[\alpha]\ c\ (y[\alpha]\ c\ n) \\
\text{map} & = & \Lambda\beta.\ \Lambda\gamma.\ \lambda f:\beta\rightarrow\gamma.\ \lambda x:\text{list}(\beta).\ \Lambda\alpha.\ \lambda c:\gamma\rightarrow\alpha\rightarrow\alpha.\ \lambda n:\alpha. \\
 & & x[\alpha]\ (\lambda z:\beta.\ \lambda r:\alpha.\ c\ (f\ z)\ r)\ n \\
\text{filter} & = & \Lambda\beta.\ \lambda p:\beta\rightarrow\text{bool}.\lambda x:\text{list}(\beta).\ \Lambda\alpha.\ \lambda c:\beta\rightarrow\alpha\rightarrow\alpha.\ \lambda n:\alpha. \\
 & & x[\alpha]\ (\lambda z:\beta.\ \lambda r:\alpha.\ (p\ z)[\alpha]\ (c\ z\ r)\ r)\ n \\
\text{reverse} & = & \Lambda\beta.\ \lambda x:\text{list}(\beta).\ \Lambda\alpha.\ \lambda c:\beta\rightarrow\alpha\rightarrow\alpha.\ \lambda n:\alpha. \\
 & & x[\alpha\rightarrow\alpha]\ (\lambda z:\beta.\ \lambda r:\alpha\rightarrow\alpha.\ \lambda w:\alpha.\ r\ (c\ z\ w))\ (\lambda w:\alpha.\ w)\ n
\end{array}$$

Binary trees are similar to lists, but with one more inductive occurrence of α in the type:

$$\begin{array}{lcl}
\text{btree}(\beta) & = & \forall\alpha.\ (\alpha\rightarrow\alpha\rightarrow\alpha)\rightarrow(\beta\rightarrow\alpha)\rightarrow\alpha \\
\text{leaf} & = & \Lambda\beta.\ \lambda x:\beta.\ \Lambda\alpha.\ \lambda n:\alpha\rightarrow\alpha\rightarrow\alpha.\ \lambda d:\beta\rightarrow\alpha.\ d\ x \\
\text{node} & = & \Lambda\beta.\ \lambda l:\text{btree}(\beta).\ \lambda r:\text{btree}(\beta).\ \Lambda\alpha.\ \lambda n:\alpha\rightarrow\alpha\rightarrow\alpha.\ \lambda d:\beta\rightarrow\alpha.\ n\ (l[\alpha]\ n\ d)\ (r[\alpha]\ n\ d) \\
\text{flat} & = & \Lambda\beta.\ \lambda x:\text{btree}(\beta).\ x[\text{list}(\alpha)]\ (\text{append}[\beta])\ (\lambda y:\beta.\ \text{cons}[\beta]\ y\ (\text{nil}[\beta]))
\end{array}$$

Vectors can be represented as functions from integers to values. The vector type consists of this function int$\rightarrow\beta$ along with an integer to indicate the size of the vector (not used here):

$$\begin{array}{lcl}
\text{vector}(\beta) & = & \forall\alpha.\ (\text{int}\rightarrow(\text{int}\rightarrow\beta)\rightarrow\alpha)\rightarrow\alpha \\
\text{newvec} & = & \Lambda\beta.\ \lambda x:\beta.\ \lambda n:\text{int}.\ \Lambda\alpha.\ \lambda c:\text{int}\rightarrow(\text{int}\rightarrow\beta)\rightarrow\alpha.\ c\ n\ (\lambda i:\text{int}.\ x) \\
\text{sub} & = & \Lambda\beta.\ \lambda x:\text{vector}(\beta).\ \lambda n:\text{int}.\ x[\beta]\ (\lambda s:\text{int}.\ \lambda v:\text{int}\rightarrow\beta.\ v\ n) \\
\text{update} & = & \Lambda\beta.\ \lambda x:\text{vector}(\beta).\ \lambda n:\text{int}.\ \lambda y:\beta.\ x[\text{vector}(\beta)] \\
 & & (\lambda s:\text{int}.\ \lambda v:\text{int}\rightarrow\beta.\ \Lambda\alpha.\ \lambda c:\text{int}\rightarrow(\text{int}\rightarrow\beta)\rightarrow\alpha.\ c\ s\ (\lambda i:\text{int}.\ \text{if}[\beta]\ (\text{eq}\ n\ i)\ y\ (v\ i)))
\end{array}$$

According to the Curry-Howard isomorphism between propositions and types, the type $\alpha \rightarrow \beta$ corresponds to the proposition $\alpha \Rightarrow \beta$, while $\forall\alpha.\, t$ to universal quantification in predicate logic. As in predicate logic, existential quantification can be defined in terms of universal quantification:

$$\exists\alpha.\, t \quad = \quad \forall\beta.\, (\forall\alpha.\, t \rightarrow \beta) \rightarrow \beta$$

Most algebraic datatypes can be modeled directly in F_2 by using their Church encodings [13, 2]. The simplest type is the type $\forall\alpha.\ \alpha$ that has no instances. The type $() = \forall\alpha.\ \alpha \rightarrow \alpha$ (also called the unit type) has exactly one instance, $()$, equal to the identity function $\Lambda\alpha.\ \lambda x : \alpha.\ x$. Booleans can be represented as follows:

$$\mathrm{bool} \quad = \quad \forall\alpha.\, \alpha \rightarrow \alpha \rightarrow \alpha$$

The first component, α, of the type corresponds to the true constructor, while the second component, α, corresponds to the false constructor. In particular, the following macro definitions:

$$\begin{array}{lcl} \mathrm{true} & = & \Lambda\alpha.\ \lambda t : \alpha.\ \lambda f : \alpha.\ t \\ \mathrm{if} & = & \Lambda\alpha.\ \lambda x : \mathrm{bool}.\ \lambda t : \alpha.\ \lambda f : \alpha.\ x[\alpha]\ t\ f \end{array}$$

define the true value and the if-then-else expression (of type $\forall\alpha.\ \mathrm{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$).

The product of the two types β and γ is expressed in F_2 as follows:

$$\mathrm{prod}(\beta, \gamma) \quad = \quad \forall\alpha.\, (\beta \rightarrow \gamma \rightarrow \alpha) \rightarrow \alpha$$

Notice that prod is a macro that takes two arguments, β and γ (i.e., β and γ are macro parameters, not type variables). (Recall that there are no user-defined type constructors in F_2.) For example, prod(bool, int) is a shorthand for $\forall\alpha.\ (\mathrm{bool} \rightarrow \mathrm{int} \rightarrow \alpha) \rightarrow \alpha$. The pair function that constructs a pair of two values $x : \beta$ and $y : \gamma$, and the functions that return the first and the second element of a pair can be coded in the following way:

$$\begin{array}{lcl} \mathrm{pair} & = & \Lambda\beta.\ \Lambda\gamma.\ \lambda x : \beta.\ \lambda y : \gamma.\ \Lambda\alpha.\ \lambda p : \beta \rightarrow \gamma \rightarrow \alpha.\ p\ x\ y \\ \mathrm{fst} & = & \Lambda\beta.\ \Lambda\gamma.\ \lambda x : \mathrm{prod}(\beta, \gamma).\ x[\beta]\ (\lambda f : \beta.\ \lambda s : \gamma.\ f) \\ \mathrm{snd} & = & \Lambda\beta.\ \Lambda\gamma.\ \lambda x : \mathrm{prod}(\beta, \gamma).\ x[\gamma]\ (\lambda f : \beta.\ \lambda s : \gamma.\ s) \end{array}$$

Sums of types (i.e., union types) can be defined in a similar way:

$$\mathrm{sum}(\beta, \gamma) \quad = \quad \forall\alpha.\, (\beta \rightarrow \alpha) \rightarrow (\gamma \rightarrow \alpha) \rightarrow \alpha$$

Integers are represented by Church numerals:

$$\begin{array}{lcl} \mathrm{int} & = & \forall\alpha.\, (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha \\ \mathrm{zero} & = & \Lambda\alpha.\ \lambda s : \alpha \rightarrow \alpha.\ \lambda z : \alpha.\ z \\ \mathrm{succ} & = & \lambda x : \mathrm{int}.\ \Lambda\alpha.\ \lambda s : \alpha \rightarrow \alpha.\ \lambda z : \alpha.\ s\ (x[\alpha]\ s\ z) \\ \mathrm{plus} & = & \lambda x : \mathrm{int}.\ \lambda y : \mathrm{int}.\ x[\mathrm{int}]\ \mathrm{succ}\ y \\ \mathrm{times} & = & \lambda x : \mathrm{int}.\ \lambda y : \mathrm{int}.\ x[\mathrm{int}]\ (\mathrm{plus}\ y)\ \mathrm{zero} \end{array}$$

The first component of the int type, $(\alpha \rightarrow \alpha)$, corresponds to the successor function while the second, α, to the zero value. The integer k is given by $\Lambda\alpha.\ \lambda s : \alpha \rightarrow \alpha.\ \lambda z : \alpha.\ s^k\ z$. That is, a

The operational meaning of programs in F_2 is defined via reduction rules that constitute the normalization algorithm of F_2:

$$\begin{array}{rcllr}
(\lambda x:t.\,e_1)\,e_2 & \Rightarrow_\beta & [e_2/x]e_1 & & \textit{(β-reduction)} \\
\lambda x:t.\,e\;x & \Rightarrow_\eta & e & \text{when } x \text{ is not free in } e & \textit{(η-reduction)} \\
(\Lambda v.\,e)[t] & \Rightarrow^t_\beta & [t/v]e & & \textit{(type β-reduction)} \\
\Lambda v.\,e[v] & \Rightarrow^t_\eta & e & \text{when } v \text{ is not free in } e & \textit{(type η-reduction)}
\end{array}$$

where $[e_2/x]e_1$ is the term obtained by replacing each free occurrence of x in e_1 by e_2 and renaming any bound variables in e_1 as necessary to prevent capture of free variables in e_2. In addition, α-conversion allows us to change the variable name of a lambda or a type abstraction:

$$\begin{array}{rcllr}
\lambda x:t.\,e & =_\alpha & \lambda y:t.\,[y/x]e & \text{when } y \text{ is not free in } e & \textit{(α-conversion)} \\
\Lambda v.\,e & =^t_\alpha & \Lambda w.\,[w/v]e & \text{when } w \text{ is not free in } e & \textit{(type α-conversion)}
\end{array}$$

It is known that all terms in F_2 are strongly normalizable (no infinite reduction sequence; i.e., all programs in F_2 terminate) and their normal form is unique (i.e., the normalization algorithm satisfies the Church-Rosser property). (See [13] for a proof.)

The domain of a type t is the set $\mathcal{D}[\![t]\!]$ given by the following two rules:

$$\mathcal{D}[\![t_1 \rightarrow t_2]\!] = \{\, f \mid \forall x \in \mathcal{D}[\![t_1]\!] :\ (f\,x) \in \mathcal{D}[\![t_2]\!] \,\} \tag{1}$$
$$\mathcal{D}[\![\forall v.\,t]\!] = \{\, x \mid \forall t' :\ x[t'] \in \mathcal{D}[\![[t'/v]t]\!] \,\} \tag{2}$$

If types are well-formed, the case $\mathcal{D}[\![v]\!]$, where v is a type variable, will never appear since every type abstraction is β-reduced in Rule 2. Note that the above two rules are not complete. For example, according to Rule 2, $\mathcal{D}[\![\forall\alpha.\,\alpha]\!] = \{\, x \mid \forall t :\ x[t] \in \mathcal{D}[\![t]\!] \,\}$. But it is well known that $\forall\alpha.\,\alpha$ has no instances, that is, its domain is empty. The actual models for F_2 are quite complex and are given elsewhere [11]. Here we are only using the above rules to prove the soundness of the typing rules.

Theorem 1 *The typing rules in Figure 1 are sound.*

Proof: The proof is straightforward if the typing environment and the type judgments are translated in the following way:

$$\begin{array}{rcl}
\mathcal{T}[\![<>]\!] & = & \text{true} \\
\mathcal{T}[\![x:t,\Gamma]\!] & = & (x \in \mathcal{D}[\![t]\!]) \Rightarrow \mathcal{T}[\![\Gamma]\!] \\
\mathcal{T}[\![\Gamma \vdash e:t]\!] & = & \mathcal{T}[\![\Gamma]\!] \Rightarrow (e \in \mathcal{D}[\![t]\!])
\end{array}$$

Then, the typing rules become tautologies. For example, the *(inst)* rule is translated into:

$$(\mathcal{T}[\![\Gamma]\!] \Rightarrow (e \in \mathcal{D}[\![\forall v.\,t_1]\!])) \Rightarrow (\mathcal{T}[\![\Gamma]\!] \Rightarrow (e[t_2] \in \mathcal{D}[\![[t_2/v]t_1]\!]))$$

which is true, if we consider the following:

$$\begin{array}{rcll}
e \in \mathcal{D}[\![\forall v.\,t_1]\!] & \equiv & \forall t' :\ e[t'] \in \mathcal{D}[\![[t'/v]t_1]\!] & \textit{(from Eq. 2)} \\
& \Rightarrow & e[t_2] \in \mathcal{D}[\![[t_2/v]t_1]\!] & \textit{(by instantiating the universal quantification)}
\end{array}$$ □

$$\Gamma \quad ::= \quad <> \quad | \quad x : t, \Gamma$$

$$\begin{array}{llll} t, t_1, t_2 & ::= & v & \text{Type variable} \\ & | & \forall v.\, t & \text{Universal quantification} \\ & | & t_1 \rightarrow t_2 & \text{Function} \end{array} \qquad \begin{array}{llll} e, e_1, e_2 & ::= & x & \text{Variable} \\ & | & e_1\ e_2 & \text{Application} \\ & | & \lambda x : t.\, e & \text{Abstraction} \\ & | & \Lambda v.\, e & \text{Type abstraction} \\ & | & e[t] & \text{Type instantiation} \end{array}$$

$$(var) \qquad \Gamma, x : t, \Gamma' \vdash x : t$$

$$(abs) \quad \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash (\lambda x : t_1.\, e) : t_1 \rightarrow t_2} \qquad (appl) \quad \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2}$$

$$(poly) \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash (\Lambda v.\, e) : \forall v.\, t} \qquad (inst) \quad \frac{\Gamma \vdash e : \forall v.\, t_1}{\Gamma \vdash (e[t_2]) : [t_2/v]t_1}$$

Figure 1: Types and terms in F_2 and the typing rules

The contribution of this paper is the extension of the type system of F_2 to support types whose instances satisfy commutativity and idempotent properties so that database collection types and their operations can be coded without inconsistencies. We present a type system that enforces the consistency of operations that involve different collection types, such as lists, bags, and sets. Finally, we give a meaning to these extensions, which is used for proving the soundness of the typing rules.

This paper is organized as follows: Section 2 gives the background for Reynold's polymorphic λ-calculus, F_2. For a better and in depth treatment of this topic, we recommend Girard's book on proofs and types [13]. Section 3 presents the F_{db} calculus and its typing rules and presents a proof sketch of the soundness of the typing rules.

2 The Second Order Polymorphic λ-calculus

Figure 1 defines the second order polymorphic λ-calculus, F_2, along with its typing rules. In addition to these rules, there are the usual rules for testing the well-formedness of types (e.g., that each type variable is bound by some universal quantification), which are omitted. We use the metavariables e for expressions, t for types, v for type variables, and x for term variables. Greek letters denote instances of type variables. The $\rightarrow$ type constructor is right associative while application is left associative. Notice the difference between the two binding operators λ and Λ: λ constructs functions from terms to terms, while Λ constructs functions from types to terms. The $\rightarrow$ represents the type of a λ, while $\forall$ (sometimes called Δ) represents the type of a Λ. A term e has type t if the type judgment $\Gamma \vdash e : t$ is derivable by the rules in Fig. 1 for some typing environment Γ.

list. Even though, all the relational operations can be easily expressed using list iterators (as it is done in [9] and [14]), these operations require that their inputs behave like bags (i.e., they are not ordered). This restriction on the input values is not syntactic, but semantic: the type system cannot tell whether the inputs to a relational operator contain duplicates or not, and therefore, it cannot detect invalid programs.

The properties that make sets and bags different from lists are the commutativity and idempotence properties. These properties cannot be enforced by the type system of F_1 alone. Consequently, we need to extend the type system of F_1 so that inconsistencies, such as converting a set into a list, are always detected during type-checking. More importantly, we need to give a meaning to these type extensions.

According to the Curry-Howard isomorphism between propositions and types, a type corresponds to a predicate and a type derivation to a proof in the predicate logic. Therefore, a type-checking system is in a way a theorem prover that derives the principal type of an expression the same way a theorem prover derives a proof. It also well-known that it is undecidable to prove whether a function is commutative and/or idempotent [3] (information required for proving whether a bag or set operation is valid). This implies that it would be undecidable as well to make such validity tests during type-checking. Consequently, the type-checking system that we describe in this paper is not complete. But, as we will demonstrate using examples, it can capture many common database operations, including all relational and nested relational operations. More importantly, our type system rejects *all* invalid operations, such as converting a set into a list. But, since our system is not complete, there would be some operations that, even though valid, are rejected by our type system.

In this paper, we introduce two new type constructors: one for commutative types and another for types that are both commutative and idempotent. Our type-checking system annotates the types that participate in these type constructors, and, during the type-checking of an expression, it propagates these annotations throughout the expression until a conflict is detected. The commutative and idempotent types have stronger annotations than the commutative types, and the latter ones have stronger annotations than the plain types. A conflict would occur whenever an annotation is found that is weaker than the one expected. If a conflict is detected, the expression being typed-checked is rejected.

Our calculus is based on the second order polymorphic λ-calculus (also called F_2), which extents F_1 by adding polymorphism via type quantification. F_2 was co-invented by Girard [13] and Reynolds [21]. It is the basis for many modern functional programming languages that support parametric polymorphism [5], such as SML [19]. We believe that F_1 is inappropriate for a database calculus for two main reasons. First, once a value has been created of type $(\text{int} \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$ for some fixed type τ, this type τ cannot be changed. Therefore, it will not be type correct to iterate over this value twice, returning values of a different type τ. There are two solutions to this problem: one is to use the untyped λ-calculus, the other is to use a universal quantification over τ, that is, to use F_2. Second, F_1 cannot capture polymorphic operations, such as the list reverse function that reverses any list of elements. The drawback is that F_2 has more complex semantics than F_1.

A Polymorphic Calculus for Database Languages[*]

Leonidas Fegaras

Abstract

We present a new database calculus, F_{db}, which is based on Reynold's polymorphic lambda calculus. It supports two new type constructors that capture the commutativity and idempotence properties of bags and sets. The type-checking rules for these extensions accept most valid programs and filter out all inconsistent ones, including for example the programs that convert bags into lists. We give the meaning of these type extensions and we prove the soundness of the type-checking rules.

1 Introduction

The simply typed λ-calculus (also called F_1), which was invented by Church [7], differs from the untyped λ-calculus in that it requires type annotations for lambda abstractions, $\lambda x : t.\ e$, where x is a parameter of type t and e is the function body. A recent work by Hillebrand et al [14, 15] uses the simply typed λ-calculus as a basis for a database language. In that language, collection types, such as lists, are represented by their Church encoding, in the same way numbers are represented by Church numerals. For example, a list of integers is represented in F_1 by a value of type

$$(\text{int} \rightarrow \tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$$

for some type τ, where the first component $(\text{int} \rightarrow \tau \rightarrow \tau)$ of this type corresponds to the type of the list constructor cons (that constructs a list given an integer and a list), while the second component τ corresponds to the type of the nil value (that constructs the empty list).

It has been shown that any recursive algebraic datatype without a contravariant recursion can be represented by a Church encoding [2]. The Church encoding of a datatype in a pure λ-calculus (i.e., a λ-calculus that does not support fix-points over types) is a higher-order type. An instance of such a type is actually a program that performs computations over this type. For example, the Church numerals are actually iteration schemes that can perform any primitive recursive computation. There are already some programming languages (e.g., LEAPS [20, 17], ADL [16], and Charity [8]) as well as some database calculi (e.g., the fold algebra [9]) that are based on iteration instead of recursion.

The main problem of representing sets and bags in F_1 using Church encodings is that they become semantically equivalent to lists. Under this representation, one can convert a bag into a

[*]*Author's address:* Department of Computer Science and Engineering, The University of Texas at Arlington, 416 Yates Street, 301 Nedderman Hall, P.O. Box 19015, Arlington, TX 76019. *Email:* `fegaras@cse.uta.edu`. *Tel:* (817) 272-3629. *Fax:* (817) 272-3784. Supported by the NSF grant IRI-9509955.