A Glimpse of Paradise

Hassan Aït-Kaci

Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92563 Rueil-Malmaison Cedex
France
email: hak@prl.dec.com

Abstract

Having understood that most attractive programming paradigms introduced recently in declarative symbolic programming languages need not be provided at the detriment of one another, we also believe that they can and should coexist with the more conventional state-effecting style of explicit control and data processing of imperative programming. To this end, we language designers must provide three primeval abstractions which could allow retrieving of most particular programming styles as particular instances, with the additional freedom of manipulating these abstractions explicitly. In the next decade, computer programming should allow specifying (1) abstract data structures, (2) abstract control structures, and (3) abstract constraint structures.

With these basic ideas in mind, we shall propose and sketch a specific design for a language and lay out some explicit requirements. Doing so, we shall try to compare this attempt to the state of the art.

> It is true, that as the form often does not accord with the intention of the art, because the material is deaf to respond, so the creature sometimes deviates from due course; ...
>
> Dante Alighieri
> *The Divine Comedy—Paradise, Canto I:121*

Motivation

It is clear that there exists a wide chasm separating the present day's "real-life" programming languages and the state of understanding of powerful high-level programming paradigms. The former provide low-level primitives and complete specific command of data and control. This results in efficiency but at the cost of prohibitive program development overhead. On the other hand, high-level programming languages offer elegant and powerful abstractions such as polymorphic typing and constraint-solving, but impose

overly restrictive programming styles often excluding the more familiar and sorely missed constructs.

Now, we believe that the application software market of the 90's will comprise as a large part programs for advanced data and knowledge representation, graphics object manipulation or reasoning thereof, natural language processing, CAD, hardware design verification, combinatorial optimization and problem-solving. These exemplify a category of programming challenges whose realization in conventional programming languages is currently very difficult and time-consuming. However, today's constraint-based languages, although providing adequate tools for specific problem domains, still confine the user to canned structures leading to unacceptable inefficiencies.

Thus, it is imperative to build a more appropriate programming environment than either side of the gap. To structure our argument, we shall word this essay as a proposal to design and build a prototypical programming environment of the 90's—dubbed *Paradise*[1]—whose building blocks will reflect our technical assessment of its required functionality. Namely, it must incorporate general and versatile abstractions to support direct execution of high-level constraint specifications coexisting along with the familiar imperative programming paraphernalia. This is bound to facilitate significantly the development of advanced applications such as we have in mind.

Our key insight is based on our having understood that most attractive programming paradigms introduced recently in declarative symbolic programming languages need not be provided at the detriment of one another. This fact is clearly demonstrated and amply illustrated in [AKN86,AKN89,AK90,AKP90,AKP91,AKM90]. More importantly, they need not preclude the more conventional state-effecting style of explicit control and data processing of imperative programming [FF89,Fel87]. Therefore, we believe that it is quite feasible to set up a concrete and rigorous design plan based on building three primeval abstractions which could allow retrieving of most particular programming styles as instances, with the additional freedom of manipulating these abstractions explicitly. Namely, programming in Paradise should allow specification of:

1. abstract data structures;
2. abstract control structures;
3. abstract constraint structures.

The first item is perhaps the most familiar and used through the machinery of modules, with added power supplied by generic modules. The second and third points are less conventional, the second one having already been under serious investigation and the last one being quite new.

With these basic ideas in mind, we can sketch a specific design for a language and lay out some explicit requirements.

The Design

We offer to design and implement Paradise as a programming system combining computational paradigms from Constraint Logic Programming [JL87,HS88], Object-Oriented Programming [GR80] and Typed Functional Programming [HMT88]. The system will

[1] Since it comes after LIFE [AK90] and could very well take a long time to be reached.

allow for a free mixture of efficiency-oriented imperative programming with specification-oriented declarative programming. At the same time, the system will be simple. Simplicity will be achieved by having only a small number of basic constructs, which are general and fit nicely together.

The language must have one uniform, record-like data structure based on classes and features. Classes are organized as an inheritance hierarchy and all operations are bound to classes. The type system supports a few built-in polymorphic classes such as pairs, lists, and arrays. Other parametric data structures can be implemented by means of generic modules. Well-typedness is checked before program execution. Execution of well-typed code is guaranteed to map well-typed memory states to well-typed memory states [Coo89]. Programs are then naturally organized as hypergraphs of possibly parametric modules. The consistency of the module structure is checked automatically.

The programming system will be interactive and incremental. A program consists of a hypergraph of modules that is created, inspected, and modified through a window and menu oriented user interface. Module and type checking as well as compilation are incremental and are done as early as possible [CCHO89]. Code that has passed the type checker can be executed immediately.

Operators can take the form of functions, relations and procedures. Operators are first-class objects. Operators are uniformly declared with a rank $\sigma \rightarrow \tau$, where σ is the type of the input argument (there is always exactly one) and τ is the type of the result. Functions do not have side-effects. Relations have monotonic side effects; that is, they can impose additional constraints on existing values. Procedures have hard side-effects; that is, they can nonmonotonically update the fields of a value.

The language is equipped with a type-safe exception handling similar to ML. However, imposition of constraints and updates of fields can be trailed, which means that upon return to a choice point (called an exception handler in ML) the initial memory state may be recovered automatically. This yields exception controlled backtracking, which is more powerful than the chronological backtracking underlying Prolog [CM84,SS86,O'K90].

One of the most innovative aspects of the language is the integration of constraint solving. On the well-understood side, the language will support unification with respect to its data structure, which amounts to sorted feature unification as realized in LogIn [AKN86]. In addition, built-in constraint solving methods are accommodated for common useful applications such as numerical problems and propositional logic. The well-founded and efficient integration of these three families of methods alone will make for an interesting subproject.

Finally, it must be made possible to program new constraint solving methods. Finding the right control primitives and the appropriate abstraction mechanisms for this task is an open research problem, solutions of which should provide for a major breakthrough in programming.

Partial Data Representation

Types are organized in a partial order hierarchy (inheritance). Data is uniformly represented as feature structures (attributed objects). This structure of objects respects type schemas (classes).

The advantage of the fact that there is no (conceptual) difference between values and types is that data may be added attributes compositionally and (variable) binding may

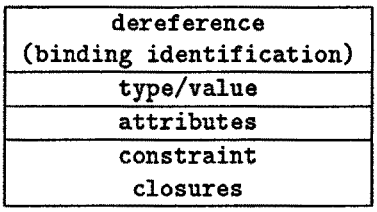

Figure 1: Partial Data Objects

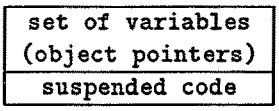

Figure 2: Constraints as Closures

be seen as object identification. This has unusual pay-offs. First, compile-time resolves some but not all types. Second, there is no need to distinguish between of `case` and `typecase` [Rov86]. Third, binding as data identification gives a natural interpretation of pointers as equivalence classes of references (coreference classes), and allows selectively undoable side-effects.

Abstract Constraint Solving

Recent work in Logic Programming has shown that Prolog's unification can be generalized to arbitrary (decidable) constraint-solving, as long as is provided an appropriate data representation and interpretation thereof, and a constraint-solver in the form of a relevant decidable, correct (sound and complete) algorithm on these structures [JL87,HS88]. This provides great possibilities to execute declarative specifications as programs.

Thus, constraint-solving is done incrementally and asynchronously. This is achieved by seeing constraint-solving as a non-deterministic normalisation process over formulae involving existential variables. Variable valuations are thus computed declaratively with a mechanism of suspended constraints as closures [AKN89,AKP91].

Hence, given a representation of objects as feature structures interpreted as partial approximations (see Fig. 1) a network of asynchronous constraints is established by representing constraints as closures (see Fig. 2). For monotonic side-effects, objects in the network are preserved by object identification realized through binding as dereference.

Abstract Control

Abstract control for programming languages in its most general form stems from research in Denotational Semantics whose computational rendition as a higher-order formalism (such as the λ-calculus) give explicit "first-class citizenship" to *continuations* [FWFD88]. Thus, Scheme [RC86] is certainly the programming language instance having explored the issue the furthest with its higher-order `call/cc` primitive which allows arbitrarily

powerful control flow. The simpler exception-handling of ML [HMT88] and Modula-2+/3 [CDG+89,Rov86], elaborations of Lisp's earlier `catch/throw` construct [M+65, Ste84], still offers some abstraction of control, although less than full continuations but much simpler to implement as mere run-stack poppers. For Paradise, we propose a compromise retaining the latter's simplicity and efficiency, yet providing a more flexible management of the run-stack and heap.

The idea is to introduce two novel control primitives: (1) generalized exception handling, and (2) event suspension. Generalized exception handling extrapolates from exception-handling such as `catch/throw` in Lisp, or `try/with` in ML and Modula2+ to handle general search strategies for variable valuations in arbitrary constraint-solving. This necessitates a selective trailing and undoing of object bindings. Higher-order continuations (*e.g.*, `call/cc` in Scheme) are inadequate here as they are meant to work in a deterministic calculus without undoing of effects. Besides, although quite powerful, they are highly unintuitive. Our solution is to extend `try/with` to allow structure-sharing and undoing of side-effects. For this, we need a (selective) "trailing" of bindings mechanism *à la* Prolog [AK91]. We can define the right primitives to get a minimal overhead while being simpler to understand and implement.

Computation suspension is a natural consequence of seeing constraint-solving as asynchronous binding propagation through a object-constraint network. Although this is to happen implicitly in most cases, a control primitive to provide an explicit handle for this is of the form `wait/until` and is inspired by Prolog-II's `freeze` [Col82]. This allows adaptive control and automatic propagation of constraints. In particular, this is how one can turn "generate-and-test" search into remarkably more efficient "test-and-generate" search.

State of the Art

Currently, there are just a few other programming systems with relatively comparable breadth of functionality: Modula-3 done at Digital's SRC [CDG+89], CHIP done at ECRC [DSvH89,vH89], CLP($\Re$) done at IBM's T.J. Watson's Research Center [JM87, JMSY88], Prolog-III developed by Prologia [Col90], and ThingLab done University of Washington [Bor81]. We shall briefly overview how similar we perceive these efforts to be as compared to Paradise's ambitions, but also express how we depart from each, yet encompassing most of their combined functionalities.

Modula-3 is an elaboration of Modula-2+ [Rov86], itself an extension of Modula-2 [Wir85]. It keeps most of Modula-2+'s features with mainly two additional capabilities: Class (single) inheritance and dynamic types. Clearly, these new features are fully part of our conception of Paradise. Paradise is expected to support (multiple) inheritance with the additional power of object unification. Furthermore, the order-sortedness of Paradise's feature terms makes no real distinction between types and values—thus achieving precisely the behavior of dynamic types. As for what it inherits from Modula-2+, Modula-3 has its traditional exception-handling mechanism—essentially non local exits, and multi-threading. On the other hand, it does not support unification nor constraint-solving (and therefore *trailing* of bindings) and side-effect unwinding exceptions. Finally, it does not offer polymorphism nor partial-objects. It does have multi-threading which we do not plan to worry about in Paradise—that is, in our current design plan.

CHIP, CLP($\Re$), and Prolog-III are all instances of Constraint Logic Programming

(CLP). They are, in fact, very similar in conception, differing only in a few specific constraint-solvers. A CLP language consists simply of an extension of Prolog with some specific constraint solving capabilities in addition to simple term unification. As for control, it relies entirely on Prolog's backtracking strategy. Thus, CHIP (Constraint Handling In Prolog) has solvers for: (1) Numeric constraints (linear equations, inequations, and disequations with rational coefficients); (2) Boolean unification (propositional logic theorem-proving); and (3) finite domains, which consist of finite discrete sets of ground values useful for mutually exclusive assignment problems. CLP($\Re$) has only (1) but over real numbers. It allows, by an implicit delay mechanism, handling of nonlinear numeric constraints, processing them only when enough variables have been instantiated to make them linear. Finally, Prolog-III has (1) and (2), as well as (infinite) rational tree unification. The latter also offers a control structure, known as *freeze*, which allows explicit delay of execution of a piece of code until some variables become instantiated.

Our main observation about the CLP class of languages is that they are all closed designs with each constraint-solver intricately wired in as opposed to being user- or library-specified. Also, control is rigidly frozen as backtracking, with the mild extension of explicit delays—but then only controlled by variable instantiation. Our proposed design naturally supersedes all this while retaining trailing and unwinding capabilities. Of course, none of these CLP languages supports object-oriented inheritance nor any imperative programming constructs.

Finally, ThingLab is yet another constraint-based programming paradigm developed at the University of Washington at Seattle. It is based on Smalltalk [GR80] and therefore makes heavy use of object-oriented style of programming. ThingLab treats constraints as networks of variables depending on one another by explicit or implicit dependencies—a super-spreadsheet. These are used to propagate values as soon as they appear as some variables' bindings, failing on conflicts. One great innovation of ThingLab is its organization of constraints into a hierarchy with different degrees of strength assigned to each level. [BDFB+87] This results in surprising flexibility for expressing default behavior of constrained systems. ThingLab also has a rich collection of strategies for controlling propagation of constraints along the network (local, decaying, *etc.*), as well as a library of, and user-specifiable, constraint-solvers.

Ideas from ThingLab (especially constraint hierarchies) are quite interesting and we are tempted, eventually, to fit some into Paradise's design. However, we have yet to work out a satisfactory formal foundation for this paradigm. On the other hand, we feel that its demon-based control mechanism, although quite powerful, is a bit too ad-hoc. The trail- and stack-effecting exception handling that we have in mind offers all the power needed with a simple and uniform semantics. Finally, we must mention intriguingly promising on-going work in the spirit of what we propose but taking directly after Thinglab [FB90].

Conclusion

We are convinced that a major momentum in programming of the 90's will be the inclusion of constraint-solving as a basic tool. Combined with the abstractions already accepted with the advent of symbolic and object-oriented programming as well as the familiar older (and irremediably useful) imperative structures, this new concept will bring significant power to the programmer. The time is ripe to start experimenting with a prototype design simply based on today's scattered and apparently uncompatible know-

how. We have sketched the basic requirements for such a ideal realization. Work is underway towards achieving it. In summary, we believe that we can already synthesize, from our experience and the current state of the art, a substantial improvement on the most advanced programming language designs—and this, without sacrificing simplicity, efficiency, nor convenience. We are truly aiming for Paradise.

References

[AK90] Hassan Aït-Kaci. An overview of LIFE. Research paper, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, 1990.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* Series on Logic Programming. MIT Press, Cambridge, MA, USA, 1991.

[AKM90] Hassan Aït-Kaci and Richard Meyer. Wild_LIFE, a user manual. PRL Technical Note 1, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, 1990.

[AKN86] Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[AKN89] Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[AKP90] Hassan Aït-Kaci and Andreas Podelski. Is there a meaning to LIFE? Research paper, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, 1990.

[AKP91] Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. Research paper, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, 1991.

[BDFB+87] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *Proceedings of the Conference on Object-Oriented Systems Languages and Applications*, pages 48–60, 1987.

[Bor81] Alan Borning. The programming language aspects of Thinglab. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[CCHO89] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proceedings of the Conference on Object-Oriented Systems Languages and Applications*, October 1989.

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mike Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research Report 52, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, USA, November 1989.

[CM84] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer Verlag, Berlin, Germany, 2nd edition, 1984.

[Col82] Alain Colmerauer. PROLOG II reference manual and theoretical model. Internal report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, October 1982.

[Col90] Alain Colmerauer. An introduction to PROLOG III. *Communications of the ACM*, pages 70–90, July 1990.

[Coo89] William Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):305–311, 1989.

[DSvH89] Mehmet Dincbas, Helmut Simonis, and Pascal van Hentenryck. Extending equation-solving and constraint-handling in logic programming. In Hassan Aït-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures, Vol. 2: Rewriting Techniques*, chapter 3, pages 87–115. Academic Press, Boston, MA, USA, 1989.

[FB90] Bjorn Freeman-Benson. Kaleidoscope: Mixing objects, constraints, and imperative programming. In *Joint Proceedings of the European Conference on Object-Oriented Programming, and the Conference on Object-Oriented Systems Languages and Applications*, pages 77–88. ACM, 1990.

[Fel87] Matthias Felleisen. *The Calculi of λ-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Indiana University, 1987.

[FF89] Matthias Felleisen and Daniel Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989.

[FWFD88] Matthias Felleisen, Mitchell Wand, Daniel Friedman, and Bruce Duba. Abstract continuations: a mathematical semantics for handling full functional jumps. In *Proceeding of the Conference on Lisp and Functional Programming*, pages 52–62, 1988.

[GR80] Adele Goldberg and David Robson. *Smalltalk 80: The Language and its Implementation.* Addison-Wesley, 1980.

[HMT88] Robert Harper, Robin Milner, and Mads Tofte. The definition of standard ML – Version 2. Report LFCS-88-62, University of Edinburgh, Edinburgh, UK, 1988.

[HS88] Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. LILOG Report 53, IWBS, IBM Deutschland, Stuttgart, Germany, October 1988. To appear in the Journal of Logic Programming.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987.

[JM87] Joxan Jaffar and Spiro Michaylov. Methodology and implemetation of a CLP system. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 196–218. MIT Press, 1987.

[JMSY88] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\Re$) language and system. Report draft, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 1988.

[M+65] John McCarthy *et al. Lisp 1.5 Programmer's Manual.* MIT Press, second edition edition, 1965.

[O'K90] Richard O'Keefe. *The Craft of Prolog.* Series on Logic Programming. MIT Press, Cambridge, MA, USA, 1990.

[RC86] John Rees and William Clinger. The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

[Rov86] Paul Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6), November 1986.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* Series on Logic Programming. MIT Press, Cambridge, MA, USA, 1986.

[Ste84] Guy Steele. *Common LISP, The Language.* Digital Press, 1984.

[vH89] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming.* Series on Logic Programming. MIT Press, Cambridge, MA, USA, 1989.

[Wir85] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, third edition edition, 1985.