Intentional Programming - Innovation in the Legacy Age

Presented at IFIP WG 2.1 meeting, June 4, 1996

Charles Simonyi
Chief Architect
Microsoft Corporation

1. The Delivery of Innovation

Even as the software industry learns to deliver higher quality programs and on more predictable schedules, we feel dissatisfied. We feel that the promise of software is so much greater thant what has been actually achieved, especially in the reuse area. We feel ashamed every time we see that on the hardware side Moore's law has worked for yet another 18 months and for yet another factor of two in performance.

There is a shortage of innovation in software, due to our limited capacity to deliver innovative abstraction mechanisms to the programmers. If we innovate in methodologies — for example by introducing structured programming[Baker], cleanroom programming[Mills], or chief-programmer teams[Baker] — we improve things, but only by some constant factor. The self-reinforcing exponential effects of automation and reuse do not come into play because the methodologies are not self-referential and are "executed" not by computers, but by human organizations of programmers.

The other delivery mechanisms for abstractions — programming languages — can be executed on computers, but have other serious limitations to their ability to spread new abstractions. First and foremost they tend to invalidate legacy code and thereby create a quandary to the end user as to when to change over to new technology. Often the pragmatic answer is "never": the user organization simply may not be able to afford the costs and manage the complexity of maintaining the old system simultaneously with the adoption and development of the new. Hence, new programming languages can take hold only under the rare conditions when due to some paradigm shift the legacy codes lose their value.

As potential hosts to abstractions, programming languages have other limitations as well. The space of acceptable notations is limited by parsing technology and the reliance on linear text streams. For example C++ is already bursting at the seams with regard to the use of parentheses in its syntax. Even if one could think of some useful abstraction, there may not be a good way to express it in the context of other C++ constructs. Lastly, domain specific knowledge is traditionally consigned to domain specific languages, so a normal system which might span several domains — such as the domains of user interface, financial calculations, strings, databases, statistics, graphics, as well as general software architecture domains —

would not be able to benefit from it. Hence large systems are written only in general purpose languages such as C, Ada, or Cobol while languages with more domain-specific features do not exist at all.

Imagine for a moment that new abstractions would not invalidate legacy code, that they would exist in an unlimited notational space, and that they could carry domain specific knowledge. What kind of abstractions would we then define and use?

> 1. Reusable abstractions. Using an abstraction multiple times is the key to productivity. Semiconductor manufacturers stamp out chips by the billions from only a few hundreds of patterns, which accounts for their unique economics. In software, unfortunately, reusability is just a design goal, not an actionable design decision. It is not unlike the desire for light weight in airplane structures. Weight is not an independent variable, it is rather a consequence of other factors, for example of the development of materials which have low weight with the other significant properties being equal. Reusability will be likewise the consequence of the development of reusable abstractions which will not lose other desirable properties in the balance. This brings us to:
>
> 2. High level abstractions. Following Fred Brooks we often talk about the division of the essential and "accidental" portions of a program. By a "higher" level of abstraction we mean an abstraction that covers the essential qualities of the program, which one would not change unless the problem statement changed, as opposed to accidental detail which one might want to change for performance optimization, for compatibility and other reasons, well within the spirit of the existing problem statement. We note that it is precisely the accidental detail which often gets in the way of reuse: two problem statements are more like to be similar than the statements and their respective detailed implementations. Furthermore, if the problem statements are not identical, but are close, we can define a yet higher level abstraction which is parameterized by the difference, and at which level reuse can be achieved. But what aspects of the system, high level or not, deserve to be abstracted? This is answered in the next point.
>
> 3. Domain specific abstractions. In our textbooks, notepads, blackboards and discussions we already use a large number of abstractions which describe the domain of discourse. For example, when discussing matrices users may define invertible, unitary, upper-triangular, or tri-diagonal matrices. When discussing user interfaces, there are menus, dialog boxes, panes, toolbars, icons, and a thousand other artifacts. Users have specific expectations of how the abstractions and the operations on them combine in the domain. As long as we can guarantee that these expectations can be fulfilled, the expression of a problem will be greatly simplified by using the existing abstractions of the domain itself. By contrast the traditional solution is to embed the user abstractions into some general purpose language abstractions (such as procedure calls, records, or classes) where the embedding itself involves accidental decisions which work against reuse, and the combinational properties of the result are those of the embedding, rather than the embedded abstractions, that is those of the programming language rather than

> of those of the domain. There can be no guarantees that the domain rules remain satisfied.

But abstraction is just one side of the program representation coin. The flip side says that there have been valid reasons for those "accidental" details in the programs and for having general purpose abstractions in the languages, namely that any high level abstraction sooner or later needed to be elaborated on a computer at which point the domain specific character of the computation will have been stripped away. We can acknowledge this need by assuring that for each abstraction there will be a corresponding "concretion"[1] which hides the implementation details, and by further assuring that the concretion process is allowed to result in any conceivable implementation. Naturally the most common criterion for the concrete code will be efficiency in space or in execution time. Other plausible criteria might be compatibility with other current components, such as the operating system, or legacy components. For example, if the operating system accepts strings that include the count of characters, the user program might choose to implement its internal strings the same way.

2. Intentional Programs

The meta-abstraction which has the properties laid down above is called the "intention". From the programmer's point of view, intentions are what would remain of a program once the accidental details, as well as the notational encoding (that is the syntax) had been factored out. Intentions express the programmer's contribution and nothing more. The name "intention" suggests that they express directly the programmer's computational intent. Once the programmer has formed an actionable (executable) thought, the programmer's next question is not "what do I have to say?", as it was the case in traditional programming, but "what I insist on saying!" that is a direct expression of the independent quantities comprised by the thought in question.

This inchoate formulation can be turned into a very specific data structure: an intentional program tree.

We start by associating an entity (a DCL node, or the declaration of the intention) with the abstract intention to give it an identity. Specific instances of the intention, such as the above thought, have their own nodes (intentional nodes), each of which points to the DCL via a "graph-like pointer". The instances may have parameters which are also nodes, which are, in turn, instances of the same or of different abstract intentions. The first instance also appears in context, as a parameter to some abstraction instance which contains it.

> At this point of our description we could say that we had an Abstract Syntax Tree (AST), where the DCLs correspond to the productions of the syntax of some programming language. This would be misleading, however, because there is no syntax and there are no productions. The DCL corresponds only to what the programmer had

[1] I am indebted to Dr. Hendrik Boom for the term.

in mind, it corresponds to an intention. But to understand precisely the distinction, we have to take a few more steps to complete the definition.

To make an intention actionable, we associate with the DCL a method which describes the semantics of the intention by specifying the process of transforming the subtree headed by the intention instance into a tree containing only primitive executable nodes with fixed semantics.

Now to obtain a runnable program, we need only traverse the intentional tree and apply the transformations indicated by the DCLs pointed to by the nodes, in a process we term “reduction”. To distinguish the transforming method from traditional object oriented methods which are to be invoked in run time, the term “extension method” or xmethod will be used. The tree containing only primitive executable nodes will be called “reduced tree” which is written in a fixed language called “R-code”. The transformations can be whimsically called “reduction enzymes”. The reduced tree is really an intermediate language for interfacing with a machine-specific code generator. The reason that we use trees for this purpose is purely an engineering decision: since reduction enzymes must already operate on trees which are their inputs, it is only natural that their outputs be in the same form also.

To be sure, the reduction need not take place in a single step, so the above description is somewhat simplified. A node may be transformed several times until the subtree is completely in R-code.

We can use xmethods to resolve other questions on intentions. How were they input? We’ll have an xmethod to help define that, whether with a command word, an icon, or a command key. How are they displayed so the program can be edited? We’ll have an xmethod for unparsing the nodes that are instances of an intention in any graphically expressible notation. In fact all behaviors of the intention in an integrated development environment can be expressed using a fixed set of xmethods, including the participation of the intention in a browser or in a source control system.

Finally, definitional closure is obtained by noting that the DCLs and the xmethods are the best represented as intention instances in their own right so that we have an uniform data structure called the Intentional Programming (IP) Tree. Since the tree contains definitions and user code at various levels of abstraction, it is fitting to introduce shared libraries as an engineering superstructure on the basic tree so that the information can be grouped and exchanged conveniently.

To accommodate constant values, there is an additional refinement to the data structure. A node may contain a block of arbitrary constant bits. The interpretation of the bits is determined by the graph-like pointer, as in any other node, so constants need not be textual either. Also, it is not required that such nodes be terminal nodes in the tree. For example, the names of declarations, as seen by the user, are stored this way.

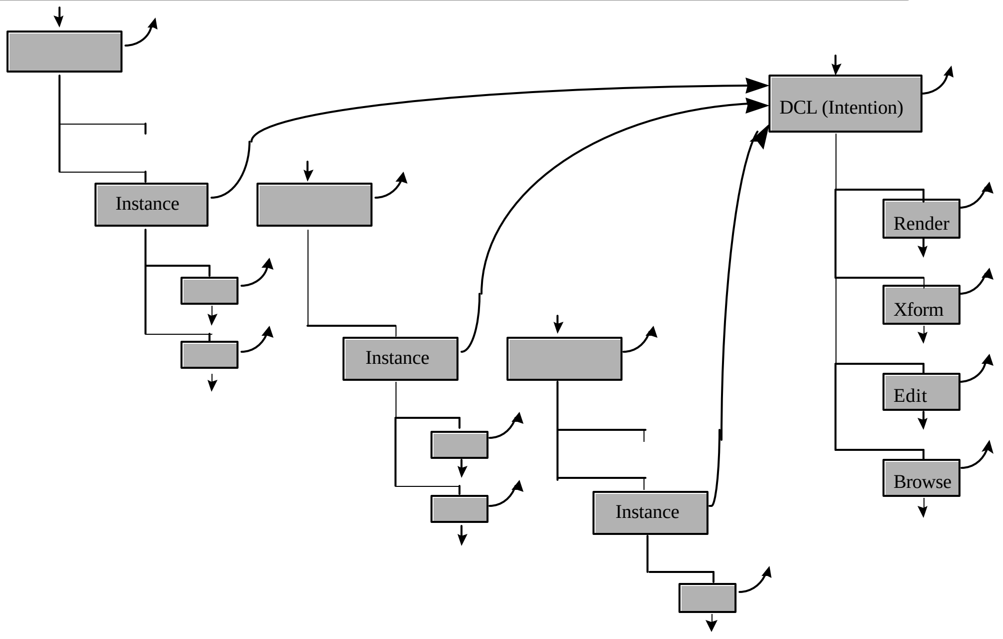


Figure 1. The Intentional Programming (IP) Tree. Curved lines represent the graph-like pointers to the definitions of the intentions, while straight lines form the tree. Every node has one parent, one definition, and zero or more offspring.

We can now check that the data structure satisfies our requirements. The graph-like pointers provide for abstraction, that is instances can refer to common parts. Intention instances are completely self-identifying with respect to their notation (syntax) and their semantics, which means that they can be arbitrarily combined and extended. Intentional nodes can be further parametrized and refined by hanging other nodes under them in the tree. New intentions can be defined without invalidating legacy code, and their combinations may follow any set of domain-specific rules. The reduction enzymes can describe any desired concretion.

Existing legacy programs can be embedded into the intentional framework by first defining the intentions comprised by the legacy language, and then adapting a legacy parser to produce the appropriate nodes instead of the usual AST. The existence of pre-processor languages, such as the one defined for C and C++, complicates the picture somewhat. The problems arise from the propensity of preprocessors to lose information such as comments — which have considerable legacy value — but also parts of “ifdefs” and the original forms of macro calls. Fortunately adequate solutions exist.

It is worth recalling what we gave up relative to computer languages. The biggest change is the abandonment of fixed syntax. The way a construct is input is no longer necessarily connected with the way it appears on the screen or in a printout. Ambiguity has also become a possibility. Two constructs may look alike but may mean different things. Since the program may not have a text representation, a general purpose text editor or, indeed, any text based tool such as a source control system or a "grep" tool can no longer be applied to programs. When stated so starkly, the changes seem gross and unjustifiable, but in fact, our emotional attachment to text and parsing is based more on technological legacy and habit than on any real benefit. More on this below.

What does it mean that the intentions can be freely combined and extended? On one hand this means invariance of notation and semantics no matter where a node may be copied to. This is different from text based languages where a piece of Snobol text "A | B" denoting a pattern alternation, when copied into a C or Pascal program will acquire completely different meanings (which are bitwise "or", or syntax error respectively). But on the other hand, the strict notational and semantic invariance is not always the most useful either. Certainly, we would want to keep the "intention" invariant while moving a piece of code into a new environment, but paradoxically this might require changes in notation and changes in semantics. One would want to change the notation so that it would match the style of the new surroundings or to try new notational conventions. The detailed semantics may have to be changed to accommodate the needs of other intentions, new global policies, such as incremental garbage collection, or updated algorithms.

The intentional tree structure promotes such changes insofar as a very large class of changes can be implemented just by updating the definitions of the intentions, of which there are fewer, while leaving the intention instances, of which there are many, untouched. This is also in contrast with text based languages, where even relatively minor changes in semantics and all changes in notation can require the editing of all expressions of an intention.

Any of the these patterns of modification can be further simplified by the creation practical inheritance schemes for xmethods which can be easily accomplished as the xmethods themselves are also implemented using intentions, that is the system is self-referential.

Changes in notation can be implemented by changing the unparser or, more practically, by organizing unparsers so that they dispatch indirect though a user-selectable "language" object which is just a style sheet for intentions. Notation is used only for interfacing with the human user, so the worst that can happen as the result of changes is that the user will be temporarily confused. Available remedies include changing back to the previous notation, or switching to a more precise notation either permanently or temporarily.

Changes in implementation are more problematic because of the potential interdependence and interplay between intentions. How can an intention representing a legacy C pointer survive in a garbage-collected environment? It cannot in the long run. However, in the short run, the garbage-collected environment can be degraded to accommodate the pointers, or a very high cost pointer implementation may be introduced that can be enumerated for garbage collections, or the memory of the system may be partitioned into fixed and garbage-collected zones with controlled access. Any of these measures could buy time for the developers to raise the intentional level of the program by eliminating the pointers, for example, while the combined system can be kept functioning. Note especially that the accommodating changes will be limited to the implementations, that is to the definitions of the intentions. We expect the number of intention instances to be so much greater than the number of intention definitions, that even the worst-case of rewriting of all transformations to achieve some modest improvement in performance or usability might make economic sense. Upgrading the legacy code, such as the removal of the pesky pointers, remains a large challenge, but maybe not as large as the alternative of starting from scratch and creating a whole new set of unimproved legacy.

Earlier we commented on how code reuse is enhanced by the use of higher level abstractions. By factoring out the implementation detail, the commonality of programs becomes more apparent and easier to exploit. Now we see that the implementation detail itself becomes a potential object of reuse. A garbage-collectable implementation of C pointers, or an implementation which detects storage leaks, is eminently shareable. In contrast to sharing the intentional code, implementation sharing occurs not between programs that are similar in functionality, but between programs with similar mixes of intentions, similar legacies, or similar space-time tradeoffs.

How does one define a new intention? First the intention is designed by straightforward listing of the independent parameters which characterize it, the parameters which the user will “insist on deciding”. These parameters may be of any form as long as they can be represented as an IP tree. A new intention will inherit a default notation which looks like a conventional procedure call. There is a default type checking enzyme which compares a type expression under the declaration of the intention with the types of the actual parameters. There is also an inherited default reduction enzyme which transforms instances into calls to a procedure specified in the intention. So a trivial intention is initially identical to a procedure call — it is displayed the same, it is type checked, and it has the same implementation. Intentions become interesting when they need parameters which are not acceptable to procedures — types, new kinds of constants, trees which describe expressions or statements — or when they affect the transformation of other intentions. To reduce interesting intentions, the designer of the intention must first decide on an implementation and then write the transformations from the source tree to the implementation. This is done not in a special-purpose fixed language, but in IP itself. Thus the transformation of IP trees can also become a domain for which special intentions can be developed. In addition to the special tree intentions, the writer of the enzyme must be familiar with the API (Application Programming Interface) that interfaces with the reducer, the program tree, and other enzymes which operate on the tree. The writer doesn’t necessarily need to understand the primitive intentions of R-Code — the reduction may create any nodes for which independent reduction already exists. If special notations desired, or if the intention interacts with other aspects of the integrated development environment, even more xmethods have to be written and additional APIs have to be used.

3. Ecology of Abstractions

IP is not a programming language. IP does not have a fixed syntax nor fixed semantics. It is an ecology for abstractions, a system where abstractions can be created, where they can survive and evolve. In the IP ecology, abstractions are the *memes*, the information carriers of the evolving ideas, in comparison with the *genes* of biology [Dawkins]. But what corresponds to the individuals of the biological ecology which serve as "survival machines" for the genes? The answer is that the user programs and programming components are the survival machines for the abstractions. The best abstractions will facilitate the widest sharing of the components they participate in, or make a program irresistible to a human user in some way, and thereby assure their own survival.

We note that programmers and users create selection pressure on components, by selecting those which are the most useful and provide feedback through various signals to other programmers who direct the evolution of the abstractions and components. Thus the ecology is complete: user programs and components co-evolve with abstractions with the programmers (and ultimately end-users) reaping the benefits.

Programming languages have always allowed for the creation and evolution of abstractions. So why is it that a dynamic ecology has not emerged before? Survival has been the missing ingredient. As we saw in Section 1, programming languages tended to kill their hosts, the legacy programs, before they delivered their benefits to a brand new collection of individuals — calling the improvements a generation would be a misnomer in our metaphor. Nonetheless, mass extinctions were avoided in some cases by concentrating on compatibility, for example as in the Fortran, Fortran II, Fortran IV, Fortran 76 series, or in the C and C++ family. This approach has been relatively successful but runs up against another phenomenon described in Section 1: abstractions, when forced into the cramped syntactic and semantic space of a programming language, tend to smother each other. IP avoids both of the traps. In fact IP promises a limited form of immortality to user programs, in that their lifetime will not be limited by the technology in which they were first expressed, nor will it be prevented from benefiting any future improvements in technology. Thus a tax program may not survive changes in the tax laws, but it can prosper through the improvements in user interface and object technologies, without rewrites.

The loose biological metaphor can be translated into a realistic economic structure. There will be several tiers of producers for abstractions, implementations, components and methodologies. There will be organizations which focus on specific domains, and create intentions and implementations for that domain. Finally, the end-user software will be created by integrating the products of the others and programming the final domain details.

It is interesting to ponder on the number of programmers who would be engaged at creating abstractions as opposed end-user programs as compared to today. In both cases the overwhelming majority of programmers (with absolute numbers in the millions) would be users rather than creators of abstractions. However the number of abstraction creators would grow dramatically from today's handful (a dozen? hundred? we are discussing successful "language designers" and "object library vendors"!) to thousands or more.

The number of different abstractions would grow accordingly. The number of domain specific abstractions would grow dramatically from a practically nonexistent base. General abstractions would also grow in number but not so steeply: they will have to compete for the finite mindshare of programmers. In this area the evolutionary pressures will push towards dramatic improvements in quality and sophistication as opposed to diversity. The number of concretions (implementations) will grow from today's one implementation per abstraction ratio to tens and perhaps hundreds to one because here individual mindshare is less of an issue.

The ability of IP to accommodate arbitrary numbers and kinds of notations, abstractions, and implementations exists just to create space for evolution. Filling up the space is not a goal or a recommendation. The correct number and kind of notations, abstractions and implementations can be determined only by the evolutionary process. The numbers might increase first, and then, in some categories, they might even decrease to pre-evolutionary levels. But the resulting artifacts will be much more complicated and adapted than those before.

4. Abstraction and Efficiency

It has been considered axiomatic that abstraction and efficiency are opposite sides of some grand tradeoff[2]. In fact they ought to be orthogonal decisions, each with its own optimization criteria. This means that an abstraction ought to be optimized for readability, modularity, modifiability, reusability, information hiding, literary value, or any other source measures, while the implementation could be as specialized, compatible, interdependent, fast, small or optimized for any other object measures as required in the instance. The separation is not difficult to achieve in an IP tree. The abstraction A is encoded as an intention declaration. The desired implementation is encoded as the output of a reduction enzyme associated with another declaration B. Finally, a connection is established between the abstraction A or just a particular instance of the abstraction, say *a*, and B. That is done by hanging an ImplAs(*b*) node under A or *a* (where *b* is just an instance of B, and parentheses denote that *b* is an offspring of the node ImplAs).

In today's computer languages efficiency is obtained either by the distribution of implementation detail over the source code or by reliance on optimizing compilers. In both of these areas heroic efforts were made by language designers and compiler implementers. On the language front, a host of abstractions, such as classes, modules, packages, templates, and inlining were offered to allow the proper hiding of implementation information within tighter boundaries. There has always been considerable competition among optimizing compilers which resulted in remarkably good code being generated from very primitive abstractions. But however hard the struggle, the results are far from optimal. If the current abstraction mechanisms were adequate, abstraction and efficiency would not be in conflict, yet they are. Years of inter-procedural optimization

[2] See for example [Coplien]: "Abstraction is often the enemy of performance. Any programmer who wants to put food on the table must balance these two."

research can not compete with what any programmer could declare at a glance: for example that a procedure has no side effects or that it is commutative.

> Granted that the automatic methods, if they existed, would be much more reliable if not absolutely reliable. The point is, however, that given the current state of the art, programmers are forced to hand-optimize code. The correctness of the optimizations may depend on the fact that a procedure has no side effects or that it is commutative, but the information will be implicit, and smeared over the code in multiple places. Instead, we could declare explicitly the key properties of procedures or other entities in order to energize the optimizer. This works because optimizers generally follow the transformational model: first an optimizable situation is recognized, then the optimizing transformation is applied. For example, if a procedure is a pure function and it is called with constant arguments, the call can be "constant folded". The hard part in this process from the optimizing software's point of view is the recognition, the easy part is the evaluation and substitution. For human programmers, it is the opposite: recognition of properties is easy and fun, while manual substitution is many orders of magnitudes slower than what can be done in software,.
>
> In IP the declarations which state invariants guaranteed by the programmer are called *stipulations*. Compared with hand-optimizations, stipulations are explicit and appear in one place only — at the place where their validity is most apparent to the reader, where they can be easily removed if suspected of being wrong or if they become invalid. It is also easy to generate run-time assertions from them for testing.

When the abstraction level is raised and domain specific abstractions are used, powerful domain specific optimizations become possible. The transformational definition of the semantics of the intentions is especially well suited to expressing optimizations.

> For example, consider the domain specific rule that Sort(Sort(X)) == Sort(X). This rule can be used to eliminate the need for sorting in the intentional statement:
>
> > Output(Sort(X))
>
> when the implementation of the collection X has been changed to a "sorted array" for some independent reason, such as speeding up statements of the sort:
>
> > x = min{n: X; n > y}; // minimum element in X which is > y.
>
> This compares with the traditional use of identities such as A+0 = A which pop up in transformations such as p->f => *(p+offset(f)).

Recall that replacing an implementation with another is a very simple operation for the programmer who uses a pre-packaged intention, so there will be very few barriers to trying new implementations. Even if an optimization, such as incremental garbage collection, requires the re-coding or modification of the reduction enzymes for most important intentions (which might number in the high hundreds) it may be well worth it for the abstraction producer, who will have a finite problem and a large market, as well as to the consuming programmer, who can verify the claimed benefits with a low cost experiment of binding the new implementations and re-reducing the source tree.

One of the most effective methods for making programs more efficient is concretion by partial evaluation, also known as specialization. This represents an intermediate point in a spectrum of possibilities between a procedure call, and inlined procedures:

> Consider the procedure F(a,b): sin(a) + 2 * b; and the calls F(x,y); F(w,v); F(z, 0); F(x, 0);
>
> The standard subroutine implementation would result in one procedure instance and four calls. If the procedure specifies that all the calls are inlined, there would be effectively four instances of the procedure body. The third and fourth instances would be simplified to sin(z) and sin(x). If partial evaluation were specified in the third and fourth instance, we would have two procedure instances and four calls. The last two calls would call the second, specialized procedure of the form F0(a): sin(a).

IP is well suited for the implementation of partial evaluation. An intention above an actual parameter may indicate that an instance of the procedure is required partially evaluated with respect to the parameters so marked. The reduction enzyme for the procedure calls can check for an existing specialization, and if there is none, partially evaluate the procedure body. Methods associated with the intentions can exploit the constant information that is distributed in the process: constant folding becomes all the more important during partial evaluation.

There is more to specialization than simply to improve speed by simplification — in fact it can never be faster than straight inlining of code. The true benefit of specialization lies in its ability to improve abstraction:

> First, procedures may be parameterized with any kinds of arguments: code, types, implementation specifications, stipulations; not just with first-class values which have run-time existence. So the power of C++ templates, Ada packages, and to a great extent C macros can be all represented by straightforward procedures. One such "argument" is the implementation definition of the type of the argument. So, for example, a library procedure Find(collection X, element e) which finds an element in a collection, when called as Find(A, a), could be automatically instantiated with respect to the implementation details of A, for example that it is a zero terminated sorted array of integers.
>
> Second, since specialization is a form of concretion, there is a specific connection between reuse and specialization. As mentioned in Section 1, related abstractions can be unified by expressing the difference as a parameter. The unified abstraction will be more reusable — if other properties, such as performance, can be kept equal. By the previous rule, differences of arbitrary kinds can be made parameters. Using specialization, a user can regenerate the specific, optimal routine from the unified abstraction. From the producer's point of view, the more reusable artifact has a better chance of creating a market that justifies the producer's investment. In other words, we predict the emergence of very flexible abstractions which have more parameters and

more kinds of parameters than what we have been used to. We term such abstractions *archetypes*.

When an archetype is used, it will be specialized with respect to almost all parameters, and further, in a given project typically there will be just one instance of its specialization. The flexibility of parameters will be there to accommodate mostly different customers or different projects, rather than different applications of the archetype in the same project.

The current conflict between abstraction and efficiency creates impossible conditions for sharing of software artifacts: if the artifact is specialized, the market will be too small. But if the artifact is general, it will not be efficient enough to sell.

5. The IP Integrated Development Environment

The IP system is illustrated in Figure 2. At the center of the picture is the single representation of the programmer's contribution: the IP tree. The programmer can enter and modify the tree using the editor. The unparsers allow the programmer to view the tree in a number of notations and formats. To execute the program, the reduction enzymes are applied and the resulting R-code is passed to the standard code generator. A source-level debugger allows the standard breakpoint and inspection operations during runtime but expressed in source terms.

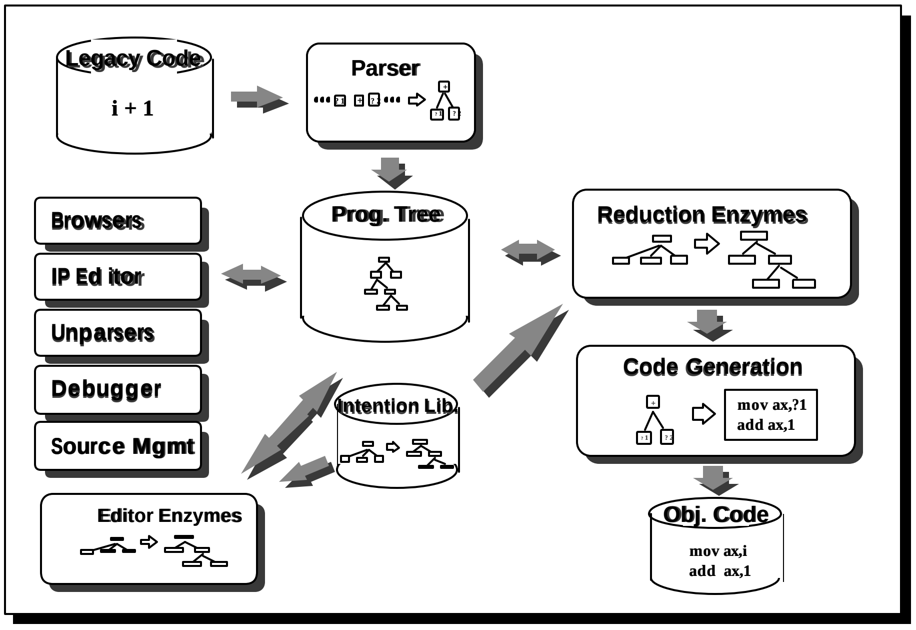


Figure 2. The IP Integrated Development Environment.

5.1. Legacy Code

Legacy code can be imported into IP by a language-specific parser. Once the code is in tree form, it is treated the same as code entered by the editor. The use of the parser depends on the inclusion of a library containing all supported intentions of the legacy language. The reading of a program into IP will not make the program more abstract, it will merely enable the combination of the program with other components and the gradual upgrading of the program to a more intentional form while it can be maintained in a functioning state.

5.2 Source Management

The source management system is necessary to support a group of programmers working on the same body of code. As noted earlier, existing text-based systems could not be used. Instead, the editor maintains a set of transactions which represents the user's changes to the tree, and the sets of transactions are checked into the shared database. To acquire the changes made by others, a merge operation can apply the checked-in transactions to the local version of the tree. During this merge, conflicts between local edits and edits made by other people can be identified. Compared to text-based systems, tree-based source managment is more difficult to implement, but it provides greater functionality. Changes are kept track of very precisely. Branch and merge operations can be added to support the development long multiple tracks. Some common merge conflicts can be automatically resolved by xmethods associated by the node. For example the renaming and the changing of the type of a declaration need not conflict, while the changing of a procedure reference and an independent change to one of the arguments of the old procedure requires user scrutiny. From the transactions, one can recreate any past version and one can inquire about the changes made in a given place in the tree.

5.3 Debugging

The greatest challenge in debugging is to be able to map the state of the program into source abstractions and source code. The latter is complicated by the transformations, but all the system needs to do is to maintain the correspondence between the source tree and the object code. So to set a breakpoint, the user makes a selection in the editor and issues the appropriate command. The selection is mapped into an object address where the breakpoint is set. To show where the program counter is in a given frame, or where an exception occurred, the procedure is reversed.

When data structures are radically altered by transformations, the system cannot be expected to develop the reverse mapping so that runtime values can be displayed in source terms. Instead, the debugger relies on an xmethod of the underlying intentional type to display values. The benefit of this approach is not only that it works even in complex situations, but also that the display can employ domain dependent and multiple formats to give the most useful feedback. For example, an index type into a collection may be shown as the numerical value followed by the element of the collection that the numerical index refers to: "11 which is the index of foo" instead of just 11.

5.4 Editing

Since the source exists only in tree form, editing in IP is done by direct manipulation through a graphical user interface. Not unlike a graphics editor, the user first selects an operand or a place in the program, and then executes a command to cut, copy, or paste something there. The command may be typed in from the keyboard, or it may be selected from a menu or an iconic toolbar. Editing actions automatically create transactions for the source control system and invalidate the object code which depends on what has been changed.

Names of quantities are important during editing, in fact it is only during editing that they are important — names are typically ignored during reduction. Without names editing would be still possible by pointing to the original to take a reference to it, and then pasting the reference at the desired place. This is good to know, because that means that it is possible to get out of any naming mess that might emerge from historical accidents. Nonetheless, the traditional concepts of scopes and other forms of name disambiguation are still valid so that the entering of quantities remains easy.

For more details on editing see Section 6.5.

5.5 Editing Enzymes

Different from the reduction enzymes, which define the semantics of the intentions, editing enzymes are used to permanently restructure the program tree. They help the programmer to re-engineer the program. They are usually but not necessarily meaning-preserving transformations.

Some editing enzymes automate common programming activities. For example, we have an enzyme to exchange the branches of an if statement and at the same time negate the condition. Another enzyme applies DeMorgan's theorem to a boolean expression. These two transformations can be applied freely when it is suspected that the resulting program might be easier to read. Another very useful enzyme takes the selected piece of code, makes a subroutine out of it, and inserts the appropriate call in its place. This enzyme takes a number of user options: where to place the resulting procedure, and how to decide which quantities are made parameters. These parameters can be specialized to one's own preferences to create a powerful command.

Editing enzymes can be easily written by a user: their API is considerably simpler than that required for reduction enzymes. For example, the portion of the DeMorgan transformation which recognizes L != R and changes it into L == R looks like this:

```
if (Pattern($$(hexprOpndL) != $$(hexprOpndR), hexpr))
      {
      RemoveTe(hexprOpndL);
      RemoveTe(hexprOpndR);
      return `($hexprOpndL == $hexprOpndR);
      }
```

The code has been made simpler by the introduction of the user defined tree-pattern intentions "Pattern", "$$" and the backquote. This is an example of the use of domain specific abstractions where the domain is the IP tree itself.

User specified enzymes are designed to be used to mechanize steps in the process of rewriting legacy programs. The enzyme can recognize a given pattern in legacy code and replace it with the intentional form.

5.6 Browsing

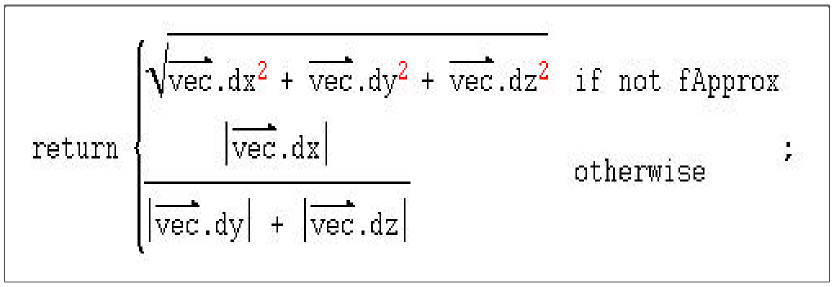
The term browsing in Integrated Development Environments refers to operations on a database of declarations in the program tree.

Because the editor operates on the tree directly, when the programmer enters, changes, or deletes some declaration, the browser information can be immediately updated.

In addition to listing declarations by various criteria, the IP browser has the capability to search for names by syllables. For example, one may type as little as OpLHex, and the syllable search will quickly determine that the name hexprOpndL was uniquely matched. This feature is important not so much to save the user some typing, but because the user frequently does not remember the precise name, but can recall key parts. The list of possibilities is displayed dynamically as the search narrows. This method is especially helpful if the names follow some consistent scheme.

5.7. Unparsers

The unparser xmethod transforms a given instance of an intention in the tree into a TEX-like string [Knuth] and additional information. These are immediately converted by IP into various data structures which support incremental updating of the windows, pointing and selection. The TEX-like encoding was necessary to allow the use of typographical notations which are typical for mathematical formulas. (See also some interesting ideas on this subject in [Abrahams]).



$$\texttt{return} \begin{cases} \sqrt{\overrightarrow{\texttt{vec}}\texttt{.dx}^2 + \overrightarrow{\texttt{vec}}\texttt{.dy}^2 + \overrightarrow{\texttt{vec}}\texttt{.dz}^2} & \texttt{if not fApprox} \\ \dfrac{\left|\overrightarrow{\texttt{vec}}\texttt{.dx}\right|}{\left|\overrightarrow{\texttt{vec}}\texttt{.dy}\right| + \left|\overrightarrow{\texttt{vec}}\texttt{.dz}\right|} & \texttt{otherwise} \end{cases} ;$$

Figure 3. Unparsing xmethods displaying the statement: return !fApprox ? sqrt(sq(vec.dx)+sq(vec.dy)+sq(vec.dz)) : abs(vec.dx) / abs(vec.dy) + abs(vec.dz);

Figure 3 shows how some normal C operators and procedures can be made to appear. The formulas are editable just as when they appear in a more traditional form. Of particular interest is the notation for vectors. The xmethods for these can come from any point in a typedef chain: maybe just the declaration

VEC vec

has the xmethod, or, more likely, the definition of the type VEC. Here is an exact copy of an unparser for the procedure for factorials which displays an exclamation point after its operand. (It is also an example of an API in need of improvement.)

```
int Factorial(int n)
      {
      int i;
      return Product(i, 1, n, i);
      }
    xmethod Render(HTE hte)
      {
      HTE rghte[1];
      CteGetRghte(hte, rghte);
      HditOutputCWrapper(hte,teprecPower,
            fTrue,fTrue,fTrue,lhNil,lhNil);
      HditOutputFormattedPsz(hte, "%h%ms", rghte[0],
            TeprecNext(teprecPower), "!");
      }
```

Conventional operator precedence is given to the API. This is necessary so that the system can automatically display parentheses at the appropriate places and it also assists an input model which follows the traditional precedences to determine what the implied operand of an operator may be. By the way, the notation with the exclamation point can be edited normally, but to enter a new factorial, the user has to type “Factorial” as was the case without the above unparser. If this is not satisfactory, the name “!” may be made to invoke a command which does the desired thing. The point is that the latter is beyond the responsibility of the unparser. Similarly, the “if expression” illustrated above is entered by typing “if”, rather than “{“ or even “?” .

6. Fears

Which brings us to a discussion of the most common fears which IP engenders.

6.1. Bad engineering is more possible than before.

Managers view the limitations of existing programming languages as control mechanisms. When their organization buys into a language, they also buy into a philosophy of limitation which effectively says: mischief caused within the language could not be helped, but at least mischief that is outside of the language has been avoided. IP raises to them the specter of unlimited mischief. But in fact, for better or worse, IP can be used for control as well as for anarchy. A transformation can generate error messages, and it can do that on the basis of domain-specific knowledge. Actionable methodologies *are* encodable into IP with their limitations. For example, if procedures without a lead-in comment are not permitted, the rule can be checked by the reduction of procedures.

6.2. How to determine what the program means?

This is a serious issue. Programmers feel that in conventional languages they can rely on their knowledge or on the documentation of a limited number of primitives to determine what any program does precisely, and hence what it means. In IP a similar procedure would involve the understanding of the reduction enzyme and the IP API used by the enzyme, which might be a finite undertaking — mathematically speaking — but daunting nonetheless. The answer is that programmers will have to generally rely on documentation of the intention rather than on inspection of the reduction enzyme to determine what the intention does, just like they used to rely on documentation to determine the semantics of language features. Special comments attached to the reduction enzyme, marked as help-file entries, may be used to prepare on-line documentation easily and to connect it with the intention. Furthermore, if all comments were removed, an intentionally encoded program should be easier to understand than one that relies on primitive semantics only. Knowing exactly what operations the machine performs, for example that it adds two integers, is still a conceptual leap away from realizing the intent: for example that the size of a union of two sets known to be disjoint is computed. In intentional form, the size would be explicitly computed on the union, with the extra knowledge of disjointness described in a stipulation and the optimization for the presence of the stipulation ensured in the reduction of the "size" intention. The "primitives" of the intentional expression are much more complicated than in the direct expression, yet they express the computational intent more fully and more invariantly under a wider range of future changes.

6.3. What will happen to legacy code? What will happen to efficiency?

See Sections 2 and 4.

6.4. Programmers will use undesirable notations.

Just like fixed semantics, the fixed notation is also lost in IP. But this is less of a problem: the specification of the notation will be typically not even a part of a program: it will belong to the individual programmer's private configuration, not unlike the desktop arrangement the choice of screensaver or sound effects. The needs of publications will require discipline on the part of the author, but not to any greater extent than, for example, what is required of mathematics papers today. People could choose any notation that the typesetter allows, but they follow conventions voluntarily to simplify communication.

6.5. Editing method is controversial.

There are three related objections to editing by direct manipulation: first, the most similar experience to direct tree editing in the past has been with Syntax Directed Editors which had a distinct lack of success. Second, programmers could not use their favorite text editors to edit programs. Third, programmers need to learn two languages: the input notation and the display notation.

Syntax-directed editing promised benefits such as the early elimination of syntax errors, the instant updating of browser database, and the ability to prompt the user with helpful information, such as the names of arguments in a statement or a procedure call. These are important benefits and the IP editor retains them. Much of the early negative experience with syntax-directed editing has been due to the lack of high quality Graphical User Interface (GUI) and also a lack of recognition of the key principles of user interface design: above all the interface should be modeless: the programmer should be able to enter programs in any order, and the temporary incompleteness of the program should not be confused with errors. For example, a reference to an undefined name need not disrupt program entry. Sure, the reference may be highlighted and entered into a “todo” list, in case the programmer forgets to create a declaration for it, or if it is a genuine typo. (The highlight color can be modulated by the closeness to existing names.) The operands of the editing operation should be clearly defined and highlighted. IP uses about half a dozen selection modes to distinguish places of insertion in various lists, operands and place for a new operator, the operators themselves without their operands, whole sub-expressions, and parts of the text which form names, constants, or comments. Editing thus acquires an intentional character: one can select the operand for a function, and then “apply”, that is insert in the tree, the function name.

The second desire, the use of one’s favorite text editor, is not something that IP can accommodate. We can only hope that the benefits provided by the various editors can be duplicated and, indeed, exceeded by the direct editing method.

Finally, the differential between the input “language” and the display notation has already been used in many applications. In graphics programs, one does not draw a circle to make a circle: one issues the MakeCircle command, whether from a menu or from an iconic toolbar. Similarly, to create a new procedure, it is actually quite efficient to type the command “proc”. As a result one gets a “complete” procedure with an empty parameter list, empty returns list, and empty body. Similarly, the command “if” inserts a new, empty if statement; to get an else clause at the outset one uses “ife”. The command “else” appends an else clause to an existing if-else list. In the scope of the integer variable *a,* the command “a” will insert a reference to that variable. In some of these instances the command is identical to the most common notation. In others, the command is the most common name, if not the notation. Consider, for example, the various notations for casts (C++ alone has two). If we do not remember them all, we *do* remember that they are notations for “casts”, otherwise how could we pose the question? To be sure, the command names are easily redefined to suit personal, organizational, or national preferences without jeopardizing in the least the information content of the programs thus created.

6.6. Source representation takes too much storage. Reduction is too slow.

Measurements indicate that the IP representation of C code is about 2.5 times the size of the ASCII source. When the 16-bit Unicode becomes prevalent, this difference will vanish altogether. It is also clear that higher level and more sharable representations will be also more compact.

Reduction speed is a problem given that commercial compilers benefited from years of performance tuning. The problem is alleviated somewhat by the ability of IP to determine for any given editing change the minimal incremental re-reductions that are necessary. The unit of recompilation may be very small, depending only on the capabilities of the available code generator.

6.7. Will there be useful new abstractions?

It is easy to find outstanding (in both meanings of the word) proposals for new abstractions [Basset][Kaelbling][Parnas][Scherlis][Shaw]. These ideas could not have been implemented because of the issues discussed in Section 1. They can be easily implemented under IP.

6.8. Enzymes will be too hard to write.

This is certainly a current problem, insofar as the API is rudimentary and the domain has not been well intentionalized yet. The long term outlook is hopeful: the tree data structure which the enzymes have to work with is conceptually very simple, and it is very simple to manipulate. Enzyme writers should get a good feel for what they need to do just by using the editor, in most cases. The domain is an excellent target for new intentions. Finally, the economic incentive for writing enzymes will be considerable and will justify the difficulties.

7. Example

The following example illustrates the way in IP to express an intention clearly, and at the same time make sure that the implementation is optimal.

The problem involves the calculation of Fibonacci numbers and the tabulation of the function values in various ways.

Let us first write down what would be considered the clearest way to express the intention. For the function, showing the recurrence relation would be probably the best expression of the computational intent:

```
 int Fib(int n)
       {
       return n < 2 ? 1 : Fib(n - 1) + Fib(n - 2);
       }
```

For tabulation, we need some enumerator constructs. There are two common styles to enumerate values in a collection: a closed loop, and an open form consisting of an initializer and a “next value” device which also tests for termination. For the closed form, we would like to write:

```
ForAll int i in Q
      printf("Fib(%u) = %u\n", i, Fib(i));
```

which could also be displayed as:

```
?  int i in Q
      printf("Fib(%u) = %u\n", i, Fib(i));
```

while the open form would be used in more complicated situations, for example when the enumeration of every second value from the collection is desired:

```
InitIter int i in Q;
while (NextIter i)
      {
      printf("Fib(%u) = %u\n", i, Fib(i));
      if (!NextIter i)
            break;
      }
```

Finally, let's choose an expression for the collection of values denoted by Q. Probably the most flexible and convenient form is to write a coroutine which produces the values in sequence. So, for example, to define the collection to contain the values {1, 2, .. 10}, we can write:

```
int ITERATOR Q
      {
      int i;
      for (i = 1; i < 10; i++)
            yield i;
      }
```

The flexibility of coroutines is evident if we want to create a more complex collection, for example the one containing the values: {-1, x, x+1, ... 10, 0}. The changes to the program would be similar to the changes in the specification, which is the sign of a well-chosen intention:

```
int ITERATOR Q1(int x)
      {
      int i;
      yield -1;
      for (i = x; i < 10; i++)
            yield i;
      yield 0;
      }
```

Up to this point we have had modest benefits from using IP. We could pick and chose the abstractions and notations from various sources: from languages such as Lisp, Clu, and standard mathematical notation.

Now it is time to specify the implementations. First of all, we know that Fib is a function, so we would want to evaluate it at compile time where possible. Next, calls to Fib from a sequential iterator should be optimized by remembering the previous values so that the cost of evaluation would be constant rather than O(n^2).

The interaction of the enumerating statements ForAll, InitIter, and NextIter with the definition of the iterator should also be optimized. For example, the closed uses of Q should be optimized to work as a normal "for" loop. The open use of Q and either use of Q1 can not be optimized this way so a more general coroutine would have to be created. The state of the coroutine — its local variables plus an "address" of the last yield — would be stored in the frame of the caller, and a pointer to this data structure will be passed as an extra argument. Q1 would be transformed into something like:

```
// TRANSFORMED CODE! NOT WRITTEN OR SEEN BY THE PROGRAMMER
bool Q1(STATEQ1 *pstateq1, int *presult)
      {
      switch (pstateq1->iyield)
            {
      case 0: goto LYield0;
      case 1: goto LYield1;
      case 2: goto LYield2;
      case 3: goto LYield3;
            }
      pstatec1->iyield = 0;
      // other initializations come here
      return 0; // return value ignored
LYield0:
// start: iyield has been initialized to 0
      *presult = -1; // yields are transformed into this
```

```
        pstateq1->iyield = 1;
        return fTrue; // not finished yet
LYield1:
        for (pstateq1->i=pstateq1->x;pstateq1->i < 10;pstateq1->i++)
              {
              *presult = psptateq1->i
              pstateq1->iyield = 2;
              return fTrue; // not finished yet
LYield2:
              }
        *presult = 0
        pstateq1->iyield = 3;
        return fTrue;
LYield3:
        return fFalse; // finished
        }
```

The declaration of the Fibonacci function is decorated with the xmethods for the recurrence transformation and the constant folding. To provide the value for constant folding, a copy of the unimproved Fibonacci code is linked with the compiler:

```
int Fib(int n)
        {
        return n < 2 ? 1 : Fib(n - 1) + Fib(n - 2);
        }
xmethod HteTransform(PMPB pmpb)
        {
        return HteRecurrenceTransform(
              /* Index of n in formals list:  */0,
                    /* dnMax:  */2,
              pmpb);
         }
xmethod FoldConstants(PMPB pmpb)
        {
        int nActual;
        if (FIntConstantArg(&nActual, pmpb))
              SetIntResult(Fib(nActual), pmpb);
        }
```

The messy details of the recurrence transformation are not given here. While the transformational solution might look like an overkill for separating the simple optimization from the specification, the same transformation may be used in many other cases, for example in the calculation of Bessel functions. This is an example of how the implementation details themselves become sharable artifacts.

The transformed main loop will look like this:

```
// TRANSFORMED CODE! NOT WRITTEN OR SEEN BY THE PROGRAMMER
int i; int FofN; int FofNM1; int FofNM2;
FofNM2=1;
printf("Fib(%u) = %u\n", 1, ForNM2);
FofNM1=2;
printf("Fib(%u) = %u\n", 2, ForNM1);
for(i=3; i<10; i++)
        {
        FofN= FofNM1 + FofNM2;
        printf("Fib(%u) = %u\n", 1, FofN);
        FofNM2= FofNM1;
        FofNM1= FofN;
        }
STATEOFQ stateofq = {0};
Q(&stateofq, &dummy);
while (Q(&stateofq, &i))
```

```
{
printf("Fib(%u) = %u\n", i, Fib(i));
if (!Q(&stateofq, &i))
      break;
}
```

This part of the program shows the expansion of the iterator into a “for” loop, the constant folding of the first two Fibonacci values, the expansion of the loop body for the initialization, and finally the substitution of the memoized variables for the recursive calls. The other call to Fib could not be optimized this way, so an instance of the unoptimized procedure is also included in the program. Similarly, the unusual open loop required an instance of Q to be generated as a coroutine similar to the Q1 example above.

The resulting code has all the tricks that we desired. Indeed if a missing trick could be identified, it could be then added to the transformations, possibly after the inclusion of some stipulations to the source code. The transformed code also shows the problems with optimal coding: intentional details are spread in multiple places. Fib has three instances (the optimized code, the unoptimized procedure and the copy in the compiler for constant folding.) Q has two instances (the “for” loop, and the transformed coroutine). These instances do not resemble each other and one of these instances, the transformed coroutine, does not resemble the intentional version very well, either. If the optimized code had to be maintained, understanding the program would be hard and mistakes would be easy to make because the hard-to-recognize consequents of Q or Fib would be easily missed.

The IP source code, on the other hand, is very easily maintained. Every decision appears in precisely one place. Just as with optimizations, if a duplication of information were to be noticed, a new abstraction that eliminates the duplication could be immediately introduced. The intentional forms can be modified in accordance with domain expectations. The notation can be changed to whatever best supports human comprehension. The intentions are not closed to future extensions. For example, the InitIter/NextIter construct can be made to work with container classes as well.

8. Acknowledgments, Status and Directions

The IP project has been the work of a large team of people. Many ideas and refinements were contributed by Will Aitken, Ted Biggerstaff, Steve Eisner, Anthony Lapadula, Paul Kwiatkowsky, Greg Kusnick, Greg Shaw, and Ramamoorthy Sridharan. We are continuing a very fruitful cooperative project with Don Batory and Yannis Smaragdakis of the University of Texas, Austin.

The first version of IP was implemented in Microsoft C, using primitive typecase-based object-oriented programming where switch statements are used to dispatch on the run-time types of objects to the appropriate methods. After considerable testing, we bootstrapped our C sources for the last time on the 1st of March, 1995. Since that day, the system has been maintained and extended entirely in itself. This allowed the introduction of xmethods and the gradual replacement of the switches by the appropriate method dispatches. Considering that we are changing the object model and that the whole system that implements the model is itself comprises about 1.7 million instances of the model, the continuing progress of the enhancements represents probably the hardest test for a legacy system upgrade under operational loads. Even the code that implements xmethods, while written under IP, had to use the simple C abstractions only, adding to the mass of legacy code which will now have to be replaced.

General usability of the system is excellent. However, the API to the xmethods leaves much to be desired and needs to be simplified and intentionalized. This is difficult to accomplish in parallel with the shift in the object model, so the project requires considerable real time for the work that needs to be done sequentially.

Many interesting problems remain, especially in reduction and in the creation of practical intention libraries. Among the reduction enzymes, the greatest problem is to schedule the application of the various enzymes, so that they all get a chance of contributing, so that they can cooperate, and so that they have the maximum freedom in what parts of the tree they can transform. The development of intention libraries is more straightforward but still necessary both to provide test cases for the xmethod API and also to simplify our day-to-day coding problems.

Our short-term goals include a complete implementation of C++ and the support of internal use of the system within Microsoft by a group different from the IP developers. Our longer-term expectations are that the IP system could be productized by the year 2000.

9. References

ABRAHAMS, P. W., Typographical Extensions for Programming Languages: Breaking out of the ASCII Straitjacket.

BACON, D. F., GRAHAM, S. L., SHARP, O. J., Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, Vol 26 No 4, December, 1994.

BAKER, F. T, Chief Programmer Team Management of Production Programming, *IBM Systems Journal*, Vol 11, No. 1, 1972

BAKER, F. T., System Quality Through Structured Programming, *1972 Fall Joint Computer Conference*

BALLANCE, R. A., GRAHAM, S. L., VAN DE VANTER, M. L., The Pan Language-Based Editing System for Integrated Development Environments. *SIGSOFT*, 1990.

BASSETT, P. G., Frame-Based Software Engineering and Iterative Design Refinement. *Software Engineering: Tools, Techniques, Practice*, April 1991.

BASSETT, P. G., Frame-Based Software Engineering. *IEEE Software*, July 1987.

BATORY, D., O'MALLEY, S., The Design and Implementation of Hierarchial Software Systems with Reusable Components. *ACM Transactions of Software Engineering and Methodology*, Vol 1, No 4.

BERLIN, L., When Objects Collide: Experiences with Reusing Multiple Hierarchies. *ECOOP/OOPLSA '90 Proceedings*, Oct 1990.

BOSWORTH, G., Objects, not classes, are the issue. *Object Magazine*, December 1992.

BURSON, S., KOTIK, G. B., MARKOSIAN, L. Z., A Program Transformation Approach to Automating Software Re-engineering. *IEEE*, 1990.

CAMERON, R. D., Efficient High-level Iteration with Accumulators. *ACM Transactions on Programming Languages and Systems*, 1989, Vol 11, No 2, pp. 194 - 211.

CHEATHAM, T.E. Jr., Reusability Through Program Transformations. *IEEE Transactions on Software Engineering*, Vol SE-10, #5.

COHEN, H. H., Source-to-Source Improvement of Recursive Programs. Ph.D. dissertation, Division of Applied Sciences, Harvard Univ., May 1980.

COPLIEN, J., After All, We Can't Ignore Efficiency. *C++ Report* Volume 8, No, 5,

DAWKINS, R The Selfish Gene, *Oxford University Press*

DEWAR, R. B. K., GRAND, A., LIU, S., SCHWARTZ, J.T., Programming by Refinement, as exemplified by the SETL Representation Sublanguage. *ACT Transactions on Programming Languages and Systems*, Vol 1, No 1, pp 27-49.

DEWAR, R. B. K., SHARIR, M., WEIXELBAUM, E., Transformational Derivation of a Garbage Collection Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol 4, No 4.

DYKES, L. R., CAMERON, R. D., Towards high-level editing in syntax-based editors. *Software Engineering Journal*, July 1990.

FEATHER, M. S., A Survey and Classification on some Program Transformation Approaches and Techniques. *ACT Transactions on Programming Languages and Systems,* Vol 13, No 3, pp. 342-371.

GRISWOLD, W. G., BOWDIDGE, R. W., Program Restructuring via Design-Level Manipulation. *Proceedings of the Workshop on Studies of Software Design,* Baltimore, May 1993.

GROGONO, P., Comments, Assertions, and Pragmas. *SIGPLAN Notices*, Vol 24, No 3.

JORDAN, M., An Extensible Programming Environment for Modula-3. *ACM*, 1990

KAELBLING, M. J., Programming Languages Should NOT Have Comment Statements. *SIGPLAN Notices*, Vol 23, No 10.

KNUTH, D. E., The TEXbook. *Addison-Wesley*, Reading Mass., 1984

KOTIK, G. B., MARKOSIAN, L. Z., Automating Software Analysis and Testing Using a Program Transformation System. *ACM*, 1989.

KOTIK, G. B., ROCKMORE, A. J., SMITH, D. R., Use of Refine For Knowledge-Based Software Development. *IEEE*, 1986.

KRUEGER, C. W., Models of Reuse in Software Engineering. Carnegie Mellon Report CS-89-188, December 1989.

MERKS, E. A. T., DYCK, J. M., CAMERON, R. D., Language Design For Program Manipulation. *IEEE Transactions on Software Engineering*, Vol 18, No 1.

MILLS, H. D., LINGER R. C., Data Structured Programming: Program Design without Arrays and Pointers. *IEEE Transactions on Software Engineering*, Vol SE-12, No 2, February 1986.

MILLS, H. D., DYER, M., LINGER R. Ccleanroom Software Engineering. *IEEE Software*, September 1987.

MINÖR, S., Interacting with structure-oriented editors. Lund University, Sweden

PARNAS, D. L., SHORE, J. E., ELLIOTT, W. D., On the Need for Fewer Restrictions in Changing Compile-Time Environments. *Naval Research Laboratory Report*, 7847.

PRIETO-DIAZ, R., Status Report: Software Reusability. *IEEE Software*, May 1993.

RIEHLE, R., Objectivism: “Class” Considered Harmful. *Communications of the ACM*, August 1992, Vol 35, No 8.

SAKKINEN, M., The Darker Side of C++ Revisited. *Structured Programming*, 13: 155-177.

SCHERLIS, W. L., Abstract Data Types, Specializations, and Program Reuse.

SHAW, M., WULF, W.A., Toward Relaxing Assumptions.in Languages and their Implementations. *SIGPLAN* 15(3), 45-61 1980

VOLPANO, D. M., KIEBURTZ, R. B., The Templates Approach to Software Use. Software Reusability, Edited by Biggerstaff, T.J., Perlis, A.J. Addison-Wesley