Facts, Resources, and the IDE/Compiler Mind-Meld

Anya Helene Bagge

Bergen Language Design Laboratory,
Dept. of Informatics, University of Bergen, Norway
http://www.ii.uib.no/~anya/

Abstract

Classical compiler architecture is centred around producing object code in a batch-oriented fashion. Integrated development environments (IDEs) present new challenges to the language implementer: information should be (almost) instantly available, presented even for incorrect code, and should be dynamically updated as the user is editing. To increase responsiveness, it may be necessary to compute information incrementally, and to make use of multicore processors. An existing compiler cannot easily be adapted to provide IDE services without impacting the user experience; typically, IDE developers have to develop their own language frontends particularly targeted at IDE use.

In this paper, we shall discuss the design of an Eclipse-integrated implementation of the experimental Magnolia programming language. The implementation is modular, with code and data being shared by both the editor and the compiler, and the design has been done with incrementality and multi-threading in mind.

Keywords: IDEs, editor-integrated compilers, incremental compilers, Eclipse, Magnolia programming language, compilation by transformation, source-to-source transformation, plugin architectures

1. Introduction

Integrated development environments (IDEs) are an essential part of a software developer's toolbox. In particular for large-scale software projects, the IDE can be an invaluable aid in navigating the code and APIs, providing an overview of the project, and doing refactorings.

Current IDEs provide a wide range of services, such as navigation, hover documentation, code completion, formatting and refactoring. To some degree, they all rely on compiler services; parsing, name resolution, analysis. Yet, traditionally, compilers have not been designed with IDEs in mind; they are focused on translation and optimisation, with access to ASTs and analysis results sometimes added as an afterthought (LLVM/Clang is a notable recent exception).

The combination of editor and compiler poses some unique challenges. The world-view is no longer static and batch-oriented; information about source code elements and errors must be maintained and updated as the user edits various code files. Data must be computed on demand, and possibly persisted across editing sessions. The IDE needs a broad view of the entire source project, so that changes in one file are reflected in its dependents.

This paper gives an overview of the IDE and compiler of the experimental Magnolia programming language [1, 2]. The IDE is built on top of Eclipse, and implements the language as a library which is shared between compiler and IDE. 'Compiler' in this sense means just the parts of the library that are used by the IDE's build system to produce code – there's no real separate compiler entity.

The IDE integration and resource management system is implemented in Java, with the Magnolia language implementation built using the Rascal meta-programming language [3, 4].

Magnolia is designed for programming based on abstraction and specification. It has a programming style that relies heavily on interfaces with integrated specifications (*concepts*), and implementation modules that can be adapted and reused in different ways (*implementations*). A typical application will make use of many different interfaces and implementation modules, and may have adapted many of them by renaming and changing declarations. For example, the definition of integers is made up of nearly thirty smaller modules that define various algebraic properties. Our experience so far makes it quite apparent that successful use of the language will require the assistance of an IDE.[1]

The contribution of this paper is a description of the the Magnolia IDE system, with a particular focus on

1. the overall design of the system (Section 2),
2. Pica, a hierarchical semi-persistent program database which maintains a workspace overview and shares information between the compiler and IDE services, and provides a subsystem for in-memory and on-disk storage of facts, based on soft references (Section 3);
3. lessons learned (Section 5.2).

We'll also briefly discuss the Magnolia implementation itself (Section 4), related systems (Section 5.1), and IDE support for language implementation (Section 5.3).

2. Design

We'll begin by giving an overview of the system design and its requirements. The main requirements for the system are as follows:

1. *Compilation:* A build system and compiler suitable for translating Magnolia code to a target language (C++).
2. *Full-featured support for IDE services:* We need a basis for implementing a wide range of editing services, from basic features like code highlighting, outlining and error/warning squigglies, to simple semantic services, including hover help and code navigation, and advanced services like refactoring and visualising code in different ways.
3. *Low implementation cost:* Because the development is done as part of a research project, aimed mainly at language design and not language tooling, implementation and maintenance should take as little effort as possible.
   'Low cost' is somewhat difficult to define; in practice, we had 1–2 person-years available to get the core implementation done; and any maintenance will have to be done without dedicated personell, on top of normal researcher duties.
4. *Flexibility:* The system needs to adapt to (rapid) changes in the language design. In particular, small language changes should result in relatively small changes to the implementation.
5. *Scalability:* The system should be able to deal with non-trival project sizes; at least tens of thousands of source lines across a large number of files.

Furthermore, the following features are desirable, but of lesser importance:

6. *IDE independence:* Ideally, we would like to be able to reuse as much of the implementation as possible if there is ever a need to support other IDEs or batch compilation.
7. *Language independence:* Considering the development effort going into this project, being able to support multiple languages makes sense. Likely additional languages would be small DSLs and language variants developed internally, or even other research languages.

The current system supports both compilation and IDE services, and has proven to be reasonably flexible when it comes to language design changes. Performance, however, is a major problem, particularly when compiling large projects. This is mainly due to trade-offs with respect to implementation cost and flexibility – it is likely that we can improve the performance dramatically by optimising or rewriting key parts of the Magnolia language implementation. We will discuss this further in Section 5.2.

[1]To a degree, the language is designed with the assumption that an IDE is available to aid the programmer. Whether this is a good idea or not remains to be seen, though it is certainly a liberating starting point for language design.

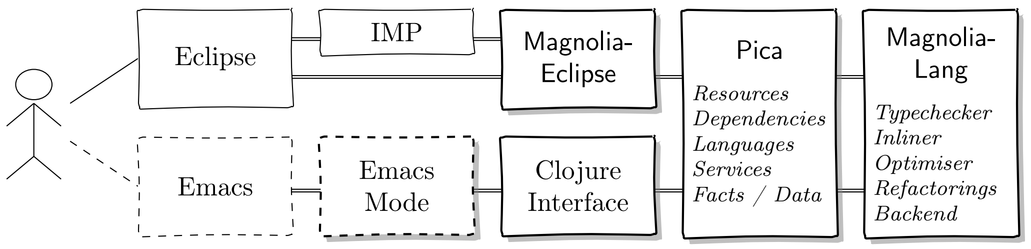


Figure 1: Architectural overview

Architecture. The system consists of the following parts:

- Pica – a language-independent resource manager and IDE service library, with support for both the Eclipse resource system and plain file systems.
- MagnoliaLang – the Magnolia language implementation library.
- MagnoliaEclipse – Magnolia IDE services for Eclipse, build on top of Pica, MagnoliaLang and IMP [5].

The overall system architecture is sketched in Fig. 1. The Eclipse-specific part, MagnoliaEclipse, is based on the IMP framework [5], and contains the glue code necessary to connect the IDE to underlying services, such as hover help, compilation and refactoring. The Pica library (described in Section 3) maintains a complete view of workspace resources and dependencies, handles in-memory and on-disk data storage, and connects to language implementation libraries (of which there is currently only one – MagnoliaLang). The language implementation library (described in Section 4.2) provides a variety of functions for manipulating, transforming and analysing Magnolia code.

Additionally, there is a Clojure interface to the system, allowing scripted or interactive access from outside Eclipse or from a different editor, such as Emacs.

The aim is to separate out Pica as a compiler and IDE support library, then build an IDE-independent *MagnoliaIDE* component which can interface to different IDEs (or even a batch compilation system), of which MagnoliaEclipse would be one.

Overall, the system has around 15 500 lines[2] of Java code and 20 000 lines of Rascal code.[3] MagnoliaLang is implemented in Rascal, while the rest is implemented in Java, with a few minor interface components written in Clojure. Clojure was chosen because of its tight integration with Java, and easy support for scripting and interactive use due to its REPL (read-eval-print-loop).

3. Pica

Pica manages a complete view of the project being edited or compiled, from files, packages and modules down to facts about individual declarations in programs. The library also keeps track of language implementations and maintains pools of evaluators for running Rascal code.

3.1. Resource Management

The Pica resource model follows that of Eclipse, where projects and source code is managed in a *workspace*. The workspace is divided into multiple *projects*, which contain files and folders. In each project, one or more folders are typically designated 'source folders' (where the compiler will look for source files), and one folder is the 'output folder' (where binaries are written). An overview of the resource hierarchy is shown in Fig. 2.

[2]As measured by David A. Wheeler's SLOCCount.

[3]By (apples-to-oranges) comparison, the Eclipse JDT, the IDE and compiler for Java, consists of around 1.8 million lines of Java code.

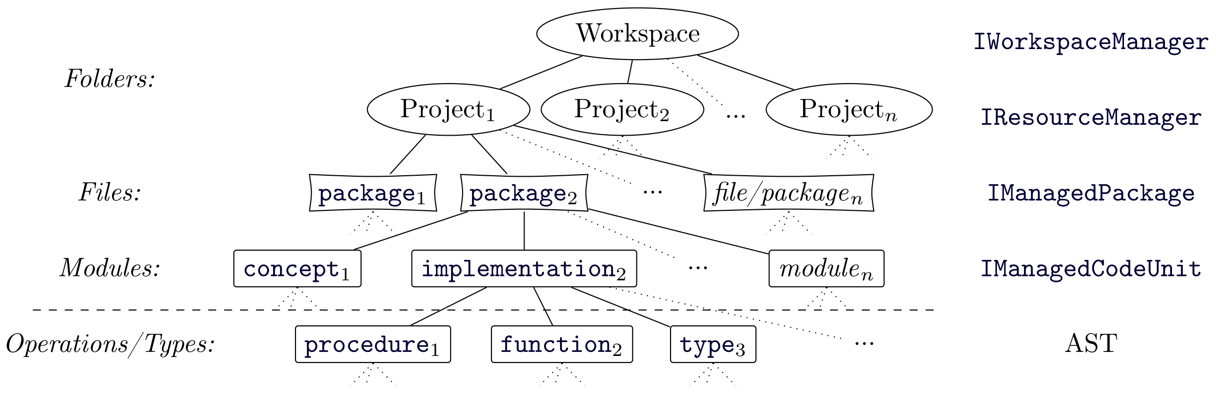


Figure 2: Hierarchical overview of the Eclipse workspace and Magnolia project organisation. *Legend:* Ellipses are purely workspace entities; boxes are language entities, with the curved boxes also being directly represented in the workspace.

Pica manages each project through a *project manager* that tracks the relevant files of that project, maintain dependencies, and manages various configuration information, such as which folders are source and output folders, and other user preferences. A *workspace manager* keeps track of the project manager, and interfaces with the underlying filesystem.

The workspace and project managers are language-independent; a separate *language registry* identifies which language a source file belongs to, so that the appropriate compiler infrastructure can be used for the file. Currently, Magnolia is the only implemented language.

Pica can interface directly with the Eclipse resource system, and get notified whenever files or resources change (e.g., when the user saves a file).

Program Resources. Pica deals with both file system resources and language-level resources. The hierarchy of language-level resources is language-specific, but we shall assume that the top-level source code entities are called *packages*. In Magnolia, a package corresponds to a file in the file system, and contains a collection of *modules*, with each module being a list of type and operation *declarations* (details are discussed in Section 4.1). In Magnolia, modules are typically quite short, so it makes sense to collect several of them in the same file (unlike Java, where a package is a folder of class files).

The Project Manager. The project manager is used by the build system to obtain the dependency graph of the project, by IDE services to map between file names and names in the programming language, and by the packages themselves to resolve dependencies. Internally, the project manager maintains a list of file resources and source code entities, and mappings from names and URIs to source entities.

The project manager presents a consistent view of all the packages in the project at a particular point in time. In an interactive editing environment, files may change at almost any time, while information about packages and dependencies may be requested at any time.

To simplify the consistency issue and reduce the amount of locking, the project manager uses an immutable data structure that can be replaced atomically. Any change notifications (files added, removed or changed) received from the workspace manager are queued for later processing. Clients that need an up-to-date view of resources must call the `processChanges` method first in order to deal with any queued changes. This is normally done by a background thread whenever resources change.

During change processing, the list of resources is updated, any changed resources are notified of the change and instructed to update themselves, and the dependency graph is recomputed. Clients can continue to access the old view of the project while the new view is computed.

3.2. Facts and Data Storage

Non-trivial information which is not directly part of the resource hierarchy is stored in *fact* objects. Facts are the smallest pieces of information that are computed and stored independently by the compiler, making them basic units of threading and incrementality. For example, package objects include facts such as the current AST; a list of children; basic information about name, kind, imports and such; an environment of all the names in the package; cross references; and information from type checking.

Each fact is associated with a signature which is computed from the fact's dependencies (i.e., the information that was used in computing the fact). The signature is used to determine whether the fact is out of date or not. When a fact is requested (e.g., due to the build system requesting the C++ code of a program module), the signature is computed and the fact's `get` method is called. The fact may be in one of three states:

1. Available, and up to date
2. Available, but out of date
3. Not available

If a fact is not in memory, it may still be available on disk, and if so it will be loaded automatically.

The signature is based on SHA hashing of source code and/or ASTs, and timestamps or version numbers of files and other facts.

If a requested fact is determined to be unavailable or out of date, it must be recomputed. When the computation is complete, the fact object is updated with the new value and signature.

In order to conserve memory, the fact objects store their values as Java soft references. A soft reference is similar to a weak reference, in that it does not count as a reference for garbage collecting purposes; i.e., if the only remaining references to an object are soft or weak references, it may be garbage collected. With a soft reference, the Java VM will try to avoid garbage collecting the object if there is still enough memory around, making soft references ideal for caching information that may be recomputed. In practice, at least for the existing Magnolia projects, the facts can be kept in memory and are only garbage collected if we artificially create a low-heap-space situation. When scaling to large code bases, however, we cannot rely on keeping everything in memory at the same time.[4] Soft references provide a simple solution to a complicated problem of figuring out what data to keep in memory.

3.3. Disk Storage

In addition to in-memory storage, the fact system also supports disk storage. Each fact object may be associated with a *store* object. Whenever the fact is updated, the new value is sent to the store. If a fact is requested but not available, the store is asked if it has the fact. Each language implementation may have its own particular storage implementation (e.g., a Java implementation could use class files), or use the default one.

The storage scheme has three components; the in-memory store objects, the on-disk store files, and the store output thread which visits the in-memory objects and writes data to disk asynchronously. Stored data will be kept in memory until the output thread has done its job, which means that it must run regularly to avoid defeating the soft-referenced fact scheme. Reading from disk happens synchronously, when the store is checked to see if it has a requested fact.

Each store may contain many facts; in the default store implementation, the file format is a simple ZIP archive, with two files for each fact; one containing the value, and one the signature.

For Magnolia, there is one store per Magnolia package file. This may be thought of as the 'compiled' version of the package, since the normal compiler output is C++ code which doesn't contain any of the information that is useful to the IDE. The file contains all the facts related to the package, and entries for all the modules of the package and their facts.

For a language such as Java where the compiler output contains all or most of the information needed by the IDE, it may make more sense to interface with the `class` files (or the non-Java equivalent).

[4]A 3–400 line Magnolia file requires around a quarter of a megabyte of memory. With, say, 10 000 files of 1 000 lines each, we're talking several gigabytes of memory.

3.4. Fine-Grained Fact Caching

The facts system allows us a certain amount of caching and incrementality, at the granularity of facts; a fact only needs to be recomputed if its dependencies have changed. This is likely sufficient to speed up a full build. For instance, a full build of the `magnolia-tests` project takes about 8s. After making a trivial change to the `Integer` package, which many of the other packages depend on, rebuilding takes about 1.4s if all affected files have to be rebuilt, and 0.4s if in-memory facts are used.

We can still do better, though. In the above example, the finest grained facts are typechecked modules, meaning that if a single function within a module is changed, the entire module needs rechecking. In an editing situation, this is something that is likely to happen often. We can reuse the same scheme, and use individual facts for typechecked functions; but this is somewhat cumbersome to implement, as much of the compiler works on Rascal trees of full modules – and this would of course not help if we later want statement- or expression-level granularity.

Rather than implement fined-grained incrementality manually, we make use of *memoisation* [6], which is a well-established technique for incremental computation [7]. Our use is inspired by Vesta [8], a system for managing and building large-scale software which caches all intermediate compilation results.

We have built a simple Rascal library that can take any Rascal function and return a memoised version of it. For successful memoisation, the function must not have global side-effects, and must not rely on values other than the arguments. This is true for most of the Rascal code in the compiler.

The memoised function is bound to a *memoisation context*, which is a hash table mapping lists of argument values (the memoisation *keys*) to results. Rascal values are always immutable, so it is safe to use them as hash table keys in this way. Furthermore, Rascal values are typically (but not always) shared, so the memory impact of keeping the argument values around is not as large as it could be, and comparison can often be done using object reference equality.

The hash table uses a fixed maximum size, with arbitrary old elements being dropped when the table is full. So far, excessive memory use has not been a problem, but we have only just started using the memoisation system. A possible memory saving technique would be to split the hash table up into soft referenced segments, that could be recycled by the garbage collector if memory gets tight.

Our memoisation scheme is meant to support fine-grained incrementality for the code the user is currently editing, while the fact system provides disk storage and coarse-grained incrementality for the whole project. Preliminary results show speedups from 1.05 for memoising simple operations on names, to nearly 2 for memoising the desugaring step of the compiler frontend. There is a certain overhead to the memoisation itself, so it is not suitable for trivial functions, or functions with a low chance of cache hits. We have (so far) no system for automatically determining which functions to memoise, the decision must be made manually.

3.5. Editing and What-If Views

So far, we have described the workspace resources mostly from the point-of-view of compilation, where the in-memory package representation corresponds to the source file stored on disk. This is not sufficient when editing code – at least not if we want an up-to-date view rather than having to save the file and wait for the builder.

To display fresh semantic information based on the contents of the editor buffer, we use an *overlay project manager*. The overlay allows us to replace the contents of some of the files with whatever is in the editor buffers. The editing systems shares a common overlay manager for all the open files. Files that are not being edited are just linked to the underlying project manager.

When an editor buffer changes, we may update the overlay, and then ask for a typecheck of the package corresponding to the buffer. This is done without changing the underlying project manager's view of the workspace. Since other packages may depend on the package(s) we are editing, they will also have to be typechecked. We can find the relevant packages by consulting the dependency graph, and then we simply copy them from the underlying manager into the overlay.

Such an overlay view is useful also in refactoring. Since it is not always possible to tell beforehand whether a refactoring will result in errors, we may do the refactoring on the overlay, typecheck it and then present the user with a diff of the changes (Eclipse has a convenient UI for this), and a list of any warnings or errors the refactoring will cause. The user can then decide whether to commit the refactoring to disk or not.

4. The Magnolia IDE

4.1. Magnolia Program Model

Magnolia code is organised hierarchically as packages, modules and declarations, collectively known as *code units* (see lower half of Fig. 2). Only the higher-level code elements are represented directly in Java and managed directly by Pica; everything from declaration level and below is represented as Rascal trees (terms, or algebraic data types) and primarily manipulated by Rascal code.

The code unit interface is split into language-independent and Magnolia-specific parts. The language-independent part includes methods for navigating the hierarchy (parent, children), and obtaining basic information (name, kind), dependency information (dependencies and dependents), various other information such as documentation, cross-references and content assist hints. The Magnolia-specific interface adds methods to obtain declaration environments and ASTs from the various functions of the compiler, such as type checking, code generation and so on.

As an example, we may ask a module for its fully-typechecked and name-resolved tree by calling the `getTypechecked()` method. Assuming no information is cached by the facts system (Section 3.2) apart from the current package, this will cause the dependency tree of the module to be computed and sorted topologically, after which the typechecked trees of all dependencies are obtained from Pica (recursively, and possibly concurrently if some modules are independent of each other). Once the dependencies have been typechecked, the MagnoliaLang is called with the set of dependencies and the unchecked tree of the module. The result is a typechecked tree (which may be stored in the facts system) and a list of errors (to be displayed in the IDE).

4.2. The MagnoliaLang Compiler Library

MagnoliaLang includes a typechecker with overload resolution, operations for renaming, combining, filtering and performing other transformations on modules, support for generating exception code from pre- and post-conditions, a prototype for performing high-level optimisations with user-defined transformation rules [9], and a backend that produces C++11 code and C++ skeleton code for interfacing with Magnolia. The compiler implementation is modular, with a separate parser and frontend, and all components implemented as a library of reusable functions.

Many parts of the library are used by both IDE services and the compiler; the frontend is perhaps the most obvious. The inliner is used both for optimisation, and as a refactoring available to the user (e.g., Fig. A.4). Even the C++ backend is used by the IDE sometimes, to create skeleton code for C++ code that should interface with Magnolia.

4.3. Eclipse IDE Services

Integration with Eclipse editing services is done through the IMP framework, which provides building blocks for standard services such as syntax highlighting, hover help, outline view, etc.

The outline view, code folding and syntax highlighting are syntax-based services, and rely only on information from the parse tree. Each code editor is linked with an IMP *parse controller*. The parse controller is notified whenever the user performs an edit, and initiates a parse of the editor buffer. The resulting parse tree is then used by IMP to provide syntax-based services. Parse errors are reported directly using error squigglies.

Other services, such as hover help and navigation require semantic information. For these cases, we rely on the Pica project manager to map the currently open file to a package object, and then query the package object for information about the particular location in the file that the user is interested in. This may trigger semantic analysis of relevant parts of the project, as sketched in Section 4.1.

Eclipse normally uses a build-on-save model, where the entire workspace is normally kept in an up-to-date, compiled state. The Magnolia builder interfaces with Eclipse and receives build requests either when manually triggered, or whenever the workspace changes (if auto build is enabled). The builder requests the entire dependency graph of the project from `Pica`, sorts it topologically, and then goes through all modules, asking either for the typechecked AST (in order to detect errors) or the compiled C++ code (for modules that are `program`s, which are the only entities for which code is generated directly).

5. Discussion

5.1. Related Systems

Background. The dream of interactive integrated programming environments goes back to the late 1970s and early 1980, with systems such as the Interlisp programming environment [10], with its (sometimes catastrophic) do-what-I-mean auto-correction; the Cornell Program Synthesizer [11]; and the PECAN system [12, 13]. The latter two are *syntax-directed* systems, where editing is controlled by the syntax of the language instead of the user entering free-form text which is later parsed – which is the norm today (with some notable exceptions, like MPS [14], and is what the Magnolia IDE does.

The *Magpie* programming environment [15, 16] is a fully incremental development environment for Pascal, built for mid-eighties single-user Tektronix workstations. Unlike PECAN and the Cornell system, Magpie relies on text input rather than being syntax-directed. It features fully incremental scanning, parsing, analysis, compilation and linking; by comparison, our system has no support for incremental parsing; and analysis and compilation are incremental only in the sense that earlier results are reused on a relatively coarse-grained basis. Magpie also has features more commonly found in dynamic languages; erroneous programs can still be run, by substituting stubs for functions or procedures that failed to compile, and programs can be updated while running by replacing / re-linking compiled code. However, 1980s workstations were not entirely up to the task of supporting such an advanced environment. The 24-bit address space of the MC68000 could support a maximum of 5000 Pascal source lines in the Magpie system; in practice, the available workstations supported only 300 lines. This is partly due to Magpie's fairly internal representation; the authors note that storing date on disk may be a way to overcome the limitation [15].

The need for scalability is a lesson to be learnt from Magpie.[5] Today, 300 line Pascal programs are not particularly interesting; Eclipse JDT easily deals with its own 1.8 million lines of code across 17000 files. An IDE has to scale, at least to the point where it can deal with realistic code bases in the target language. Incrementality can help with execution time scalability; the total size of your project doesn't matter much if you only need to process the most recently edited function. Incrementality was fairly a standard in the 1980s; today's fast processors have made this less necessary.

Current Systems. Today, the state-of-the-art in IDEs is represented by systems such as IntelliJ IDEA, NetBeans, Eclipse and Visual Studio, of which Eclipse seems to have the largest mindshare in the academic tooling community. The core of Eclipse is a plugin framework; every interesting feature is actually a plugin in the system – including the popular Eclipse Java Development Tools (JDT).

Development of new language plugins for Eclipse used to be quite an undertaking. A classic strategy is to simply clone the JDT and make modifications as necessary [17]. More recently, there are language independent frameworks for creating Eclipse language plugins; the Dynamic Language Toolkit for dynamic languages, and the general framework IMP [5], which forms the basis for the Magnolia plugin, and is also used in language workbenches like Spoofax [18] and Rascal.

The IMP framework provides a wide range of services that can be customised to fit a particular language. Our system extends and to some degree complements IMP, but replaces most of the system for handling analyses and the services that rely on them. In IMP, each open editor maintains a source model (some form of AST). Some services, such as syntax highlighting, are triggered by changes to the model. Other services, such as hover help and quick fixes, are triggered by the user. Each service may require particular analyses; for example, outlining requires parsing, and hover help requires type checking and name binding.

[5]In addition to the fact that nearly every interesting thing has already been done in the 1980s.

IMP schedules analyses as needed, either eagerly or on demand. When triggered, a service is given an AST to work on – either directly from the parser, or as the result of some analysis.

Compared to IMP, our approach is, in essence, centered around a project-wide semi-persistent hierarchical database, rather than a per-editor AST to which various analyses are applied. The information is shared between the compiler and the IDE services (the compiler is just another service). Our system is also less tightly-integrated with Eclipse. The choice of a database-like design is influenced by the Magnolia language design, where there is a large degree of interdependencies between files, and where files do not correspond to compilation units. Avoiding too-tight Eclipse integration is necessary in order to avoid locking the language implementation to the Eclipse platform.

5.2. Evaluation and Current Status

Looking back at the requirements of Section 2, the system currently provides an incremental build system and a Magnolia-to-C++ compiler. The basis for IDE services is in place, though many of the services themselves are not yet implemented. Available services include hover help (see Fig. A.5), error markings, navigation, outlining, code folding, dependency graph visualisation (see Fig. A.3), and an inline refactoring (see Fig. A.4).

Performance is a problem; compiling a small project can take minutes, and wait time for the hover helper may be several seconds if information needs to be recomputed. If we have achieved scalability, it is in the sense that the system performs equally poorly on small and large projects.

However, the performance problem is mainly in the MagnoliaLang part of the system, and is most likely linked to the performance of the underlying Rascal language. For a proper evaluation of the performance of Pica subsystem itself, we should interface with a fast language implementation.

We have gone through several major changes to the language design, and have encountered no particular difficulties in updating the Rascal-based language implementation.

Regarding implementation cost, the described system has been implemented over a period of three years, with between 25%–75% of a programmer dedicated to it. Development of the Rascal code for typechecking and code generation has been fairly straight-forward, with largest part of the development effort going to the Eclipse integration, the Pica resource and fact management system and the IDE services.

In particular, several iterations of the resource and fact system have been done, with an earlier attempt at fine-grained dependency tracking with change notifications and lazy evaluation being rejected because it turned out to be too complicated in practice. The current implementation is quite simple and straight-forward.

We have little basis for evaluating language independence yet, as we have not yet attempted to fit another language into the system. The facts and resources system should be reusable, but it relies to some degree on having a clean module system. In particular, languages like C and C++, with include-style “modules” and lexical macros will likely not fit very well.

Concerning IDE-independence, we have an experimental Emacs mode, and the necessary interfaces are in place so that we may run the system independently of Eclipse, and query the system through a Clojure interface. This has, however, only been used briefly for testing purposes.

Technology Choices. The system is implemented in Java and Rascal. As the Eclipse platform is built on Java, the language was a natural (and necessary) choice for the parts of the code that interfaces with Eclipse.

Nearly all the code-manipulation code is written in Rascal. This is a task for which a dedicated meta-programming language seems well suited; in particular, lack of features such as pattern matching and generic traverals makes Java cumbersome for this sort of task.

Rascal has a Java-like syntax, runs on top of and integrates with Java, and has a feature set well-suited for meta-programming: built-in syntax formalism and generalised parser generator; pattern matching and pattern-based function call dispatch; support for concrete syntax patterns; a wide range of immutable data types (but with mutable variables), including trees, sets, relations and associated operations; and powerful comprehension and generic traversal operators.

Rascal was chosen based on its compelling feature set and usability, and on the desire to try something new – perhaps not the best of reasons, but hopefully excusable in an academic setting. Overall, Rascal has proven quite effective for its task, though there have been problems with performance and with stability of both the language and its implementation – as is probably to be expected when developing an experimental prototype of an experimental language using an experimental meta-programming language. As such, using well-established tooling like Stratego or ANTLR might have been a better choice. On the other hand, trying out Rascal on a large language implementation task has allowed us to stress-test the design and implementation of Rascal, which is a useful endeavour in itself.

IMP has left us with mixed feelings. On the one hand, it is very easy to set up a fairly slick editor with basic IDE services; but, on the other hand, it can be frustratingly difficult to do something any other way than the 'IMP way'. Also, while the getting-started documentation is good, in-depth documentation is sparse. We are unsure how much more work it would be to work directly with Eclipse.

5.3. IDE Support for Language Implementation

Integration of a compiler into an IDE brings advantages also to the compiler writer. In particular, the IDE may be leveraged in the debugging and construction of the compiler itself, for example by visualising and allowing navigation or even interaction with internal data structures – such as the AST, dependency graph, analysis results and so on.

The Eclipse integration of the Soot compiler framework [19] allows the results of Java code analysis to be presented in the Java editor, using colours, links and text tags. This is useful both in debugging the analysis, and in using the analysis to understand code. A particular challenge in this integration is the communication of data back from the external Soot framework, especially since Soot operates on a low-level IR not directly linked to Eclipse's Java representation. This would be no problem in our case, where the compiler infrastructure is integrated into the IDE already.

The Spoofax language workbench [18] is a good example of the advantages of IDE-integrated language implementation. Spoofax allows the IDE plugin for a language to be programmed and used in the same IDE instance, with changes to the implementation reflected in open editors for that language. At any time, the developer might select some code, and view the AST for that code, apply transformations to it, and so on. IDE editing services can be controlled through code written in small domain-specific languages, and compiler, analysis and transformation services can be implemented in the Stratego transformation language [20].

So far, our system provides only rudimentary support for debugging itself, in the form of various commands for dumping some internal information in a text buffer (overview of resources, typechecked AST). Further debugging support is created as needed in an ad-hoc manner, and typically discarded afterwards. We did however employ visualisation in an earlier version of the system, where we experimented with automated fact dependency tracking and lazy recomputation of facts. The fact dependencies and the effect of triggering changes were shown using the Rascal visualisation framework [21]. This could easily be applied to visualise dependencies between modules, or even call graphs and so on. This should make interesting student projects for the future.

6. Conclusion

Taking advantage of the integration of editor and compiler requires maintaining an overview of the user's workspace, and the facts produced by various analyses and compiler faces. The Magnolia IDE uses a central program database, which maintains both a hierarchical workspace overview, a dependency graph and a soft-referenced, persistable database of facts as well as a memoisation cache.

The system is used to support the Magnolia programming language; both the language and the system itself should be considered experimental.

Acknowledgements.

Some of this work was done while the author was visiting the SWAT team at CWI in Amsterdam. The research has been financed by the Research Council of Norway. Thanks to Tero Hasu and Andreas Hjortland, who have contributed to the development of the system described in this paper. And, many thanks to the WASDeTT reviewers very constructive and useful reviews.

References

[1] A. H. Bagge, Constructs & concepts: Language design for flexibility and reliability, Ph.D. thesis, Research School in Information and Communication Technology, Department of Informatics, University of Bergen, Norway (2009).

[2] A. H. Bagge, M. Haveraaen, Interfacing concepts: Why declaration style shouldn't matter, in: T. Ekman, J. J. Vinju (Eds.), Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09), Vol. 253 of Electronic Notes in Theoretical Computer Science, Elsevier, York, UK, 2010, pp. 37–50. doi:10.1016/j.entcs.2010.08.030.

[3] P. Klint, T. van der Storm, J. Vinju, Rascal: A domain specific language for source code analysis and manipulation, in: SCAM '09: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, IEEE Computer Society, Washington, DC, USA, 2009, pp. 168–177. doi:10.1109/SCAM.2009.28.

[4] P. Klint, T. van der Storm, J. Vinju, Easy meta-programming with Rascal, in: J. Fernandes, R. Lämmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering III (GTTSE'09), Vol. 6491 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 222–289. doi:10.1007/978-3-642-18023-1_6.

[5] P. Charles, R. M. Fuhrer, S. M. Sutton, Jr., E. Duesterwald, J. Vinju, Accelerating the creation of customized, language-specific IDEs in Eclipse, in: Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, OOPSLA '09, ACM, New York, NY, USA, 2009, pp. 191–206. doi:10.1145/1640089.1640104.

[6] D. Michie, "Memo" functions and machine learning, Nature 218 (1968) 19–22. doi:10.1038/218019a0.

[7] W. Pugh, T. Teitelbaum, Incremental computation via function caching, in: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '89, ACM, New York, NY, USA, 1989, pp. 315–328. doi:10.1145/75277.75305.

[8] A. Heydon, R. Levin, Y. Yu, Caching function calls using precise dependencies, in: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, PLDI '00, ACM, New York, NY, USA, 2000, pp. 311–320. doi:10.1145/349299.349341.

[9] A. H. Bagge, M. Haveraaen, Axiom-based transformations: Optimisation and testing, in: J. J. Vinju, A. Johnstone (Eds.), Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008), Vol. 238 of Electronic Notes in Theoretical Computer Science, Elsevier, Budapest, Hungary, 2009, pp. 17–33. doi:10.1016/j.entcs.2009.09.038.

[10] W. Teitelman, L. Masinter, The interlisp programming environment, Computer 14 (1981) 25–33. doi:10.1109/C-M.1981.220410.

[11] T. Teitelbaum, T. Reps, The cornell program synthesizer: a syntax-directed programming environment, Commun. ACM 24 (9) (1981) 563–573. doi:10.1145/358746.358755.

[12] S. P. Reiss, Graphical program development with PECAN program development systems, in: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, SDE 1, ACM, New York, NY, USA, 1984, pp. 30–41. doi:10.1145/800020.808246.

[13] S. P. Reiss, An approach to incremental compilation, in: Proceedings of the 1984 SIGPLAN symposium on Compiler construction, SIGPLAN '84, ACM, New York, NY, USA, 1984, pp. 144–156. doi:10.1145/502874.502889.

[14] JetBrains, MPS – Meta Programming System.
URL http://www.jetbrains.com/mps/

[15] M. D. Schwartz, N. M. Delisle, V. S. Begwani, Incremental compilation in magpie, in: Proceedings of the 1984 SIGPLAN symposium on Compiler construction, SIGPLAN '84, ACM, New York, NY, USA, 1984, pp. 122–131. doi:10.1145/502874.502887.

[16] N. M. Delisle, D. E. Menicosy, M. D. Schwartz, Viewing a programming environment as a single tool, in: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, SDE 1, ACM, New York, NY, USA, 1984, pp. 49–56. doi:10.1145/800020.808248.

[17] S. V. Gomanyuk, An approach to creating development environments for a wide class of programming languages, Programming and Computer Software 34 (4) (2008) 225–236. doi:10.1134/S0361768808040063.

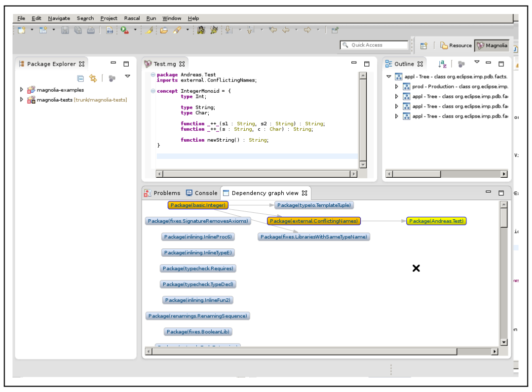
[18] L. C. L. Kats, E. Visser, The Spoofax language workbench. Rules for declarative specification of languages and IDEs, in: M. Rinard (Ed.), Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA, 2010, pp. 444–463. doi:10.1145/1869459.1869497.

[19] J. Lhoták, O. Lhoták, L. Hendren, Integrating the Soot compiler infrastructure into an IDE, in: E. Duesterwald (Ed.), Compiler Construction, Vol. 2985 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2004, pp. 2725–2725. doi:10.1007/978-3-540-24723-4_19.

[20] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A language and toolset for program transformation, Sci. Comput. Program. 72 (1-2) (2008) 52–70. doi:10.1016/j.scico.2007.11.003.

[21] P. Klint, B. Lisser, A. van der Ploeg, Towards a one-stop-shop for analysis, transformation and visualization of software, in: A. Sloane, U. Aßmann (Eds.), Software Language Engineering, Vol. 6940 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2012, pp. 1–18. doi:10.1007/978-3-642-28830-2_1.

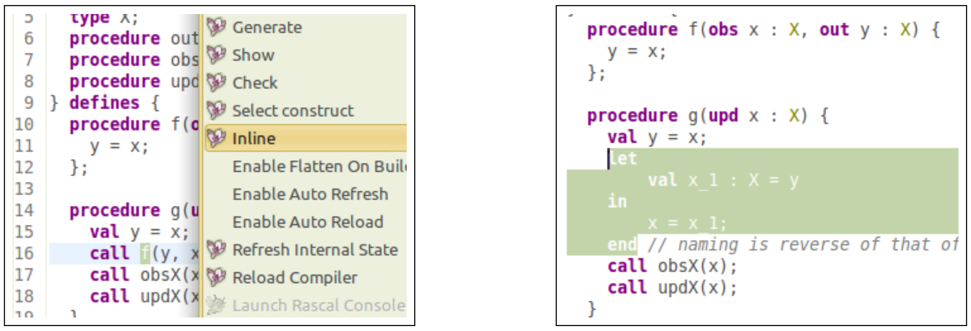
Appendix A. Screenshots



```
package Andreas.Test
imports external.ConflictingNames;

concept IntegerMonoid = {
        type Int;

        type String;
        type Char;

        function _++_(s1 : String, s2 : String) : String;
        function _++_(s : String, c : Char) : String;

        function newString() : String;
}
```

File Edit Navigate Search Project Rascal Run Window Help
Quick Access
Resource
Magnolia
Package Explorer
magnolia-examples
magnolia-tests [trunk/magnolia-tests]
Test.mg
Outline
appl - Tree - class org.eclipse.imp.pdb.facts
prod - Production - class org.eclipse.imp.
appl - Tree - class org.eclipse.imp.pdb.fa
appl - Tree - class org.eclipse.imp.pdb.fa
appl - Tree - class org.eclipse.imp.pdb.fa
Problems
Console
Dependency graph view
Package(basic.Integer)
Package(typeIo.TemplateTuple)
Package(fixes.SignatureRemovesAxioms)
Package(external.ConflictingNames)
Package(Andreas.Test)
Package(inlining.InlineProc6)
Package(fixes.LibrariesWithSameTypeName)
Package(inlining.InlineTypeE)
Package(typecheck.Requires)
Package(typecheck.TypeDecl)
Package(inlining.InlineFun2)
Package(renamings.RenamingSequence)
Package(fixes.BooleanLib)

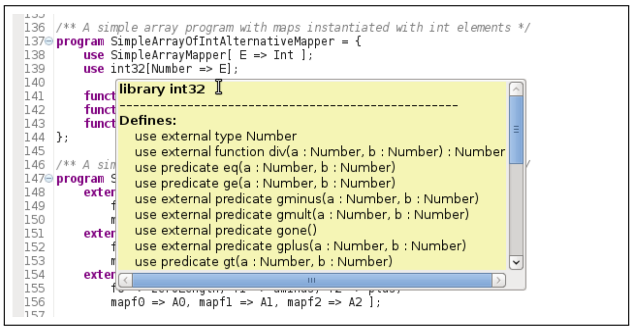
Figure A.3: IDE view with dependency graph visualisation.



```
    type X;
    procedure out
    procedure obs
    procedure upd
} defines {
    procedure f(o
        y = x;
    };

    procedure g(u
        val y = x;
        call f(y, x
        call obsX(x
        call updX(x
```

```
procedure f(obs x : X, out y : X) {
    y = x;
};

procedure g(upd x : X) {
    val y = x;
    let
        val x_1 : X = y
    in
        x = x_1;
    end // naming is reverse of that of
    call obsX(x);
    call updX(x);
}
```

Figure A.4: Selecting the *inline* refactoring, and inlining the call to `f`



```
/** A simple array program with maps instantiated with int elements */
program SimpleArrayOfIntAlternativeMapper = {
    use SimpleArrayMapper[ E => Int ];
    use int32[Number => E];

    funct
    funct
    funct
};

/** A sin
program S
    exter
        f
        m
    exter
        f
        m
    exter
        f
        mapf0 => A0, mapf1 => A1, mapf2 => A2 ];
```

```
library int32
--------------------------------------------------
Defines:
    use external type Number
    use external function div(a : Number, b : Number) : Number
    use predicate eq(a : Number, b : Number)
    use predicate ge(a : Number, b : Number)
    use external predicate gminus(a : Number, b : Number)
    use external predicate gmult(a : Number, b : Number)
    use external predicate gone()
    use external predicate gplus(a : Number, b : Number)
    use predicate gt(a : Number, b : Number)
```

Figure A.5: The hover helper is essential in understanding Magnolia code.