Programming Language Technology for Niche Platforms

Tero Hasu

Thesis for the degree of philosophiae doctor (PhD)
at the University of Bergen

2017

Date of defence: 3 March 2017

To my parents

Preface

Moulding and composing the pieces of this book has been quite a venture, with plenty of challenges, joys, and lessons along the way. I suspect the ideas here will come to influence my personal software development practices going forward. Despite the niche appeal of the subject matter, it's my hope that I'm not the only one to find some inspiration from within these pages.

Bergen, 6 October 2016

Acknowledgements

I'm grateful to my supervisors Magne Haveraaen and Anya Helene Bagge for the opportunity to come to Bergen in the first place, and for the supervision, of course. Magne's ideas on domain engineering in particular have influenced me; I was quite unfamiliar with the discipline back when I arrived. Anya's papers had struck a chord with me long before that, and were the reason why I knew of BLDL at all. Anya also provided me with the LaTeX sources of her dissertation to use as a template, and that has influenced the organization of this text.

The Research Council of Norway granted me a scholarship through the Design of a Mouldable Programming Language (DMPL) project. Such a "license" to do full-time PL research was of great personal significance to me.

The Department of Informatics at the University of Bergen has been a welcoming and pleasant workplace. I've enjoyed discussing topics of the trade with my ex-colleague Eva Burrows, my "predecessor" Valentin David, and a more recent BLDL reinforcement Jaakko Järvi. May-Lill Bagge has been almost BLDL personnel as well, and a co-traveler and conspirator on many an occasion. Ida Rosenlund, Tor Bastiansen, and other administrative personnel at the department made it so that I didn't need to be worried about university bureaucracy.

It was a pleasure to hang out by the water cooler and elsewhere with Atle, Bo, Eivind, Ina, Kristoffer, Mattia, Paolo, Puja, Samson, Sara, and the rest of the varied Bergen crew.

Matthew Flatt graciously hosted my visit to Utah. The visit was a memorable occasion in being my first experience of coworking and collaborating with others who program in Racket, despite my discovery of the language years before. Resident Racketeers and Schemers in Utah included Xiangqi Li, Eric Holk, and Andy Keep.

Prior to coming to Bergen I was at the Helsinki Institute for Information Technology HIIT, and when it comes to my career move to pursue a PhD in

programming languages, Antti Oulasvirta and Ken Rimey in particular helped make it happen: Antti supported my seeking of an external scholarship; and Ken helped me get started with PL research in the first place, in two of his projects. Antti Ylä-Jääski facilitated my earlier explorations of postgraduate study possibilities at Aalto University.

Lili could be trusted to help me relax and regain some sanity whenever I'd emerge from immersive coding and writing sessions.

Scientific Environment

The research presented in this dissertation has been conducted at the Bergen Language Design Laboratory in the Programming Theory group of the Department of Informatics at the University of Bergen, and while visiting the PLT group at the School of Computing at the University of Utah.

ICT

Research School in
Information and Communication Technology

UNIVERSITY OF BERGEN
Department of Informatics

Abstract

Developers writing software for a niche platform are denied the luxury of a first-class vendor-supported integrated development environment and a large community crafting platform-tailored libraries, tools, and documentation. I outline a strategy for setting up a cross-platform software product line with cost-effective targeting of niche platforms in mind.

The product line setup strategy assumes little tool support from the platform vendor or third parties, instead relying on a suitably-designed, malleable general-purpose programming language for the necessary support. The required language support includes: program translation into the relevant vendor-favored languages; human-comprehensible translator output to allow for basic debugging irrespective of available tools; a component system for managing software assets and assembling products; static reasoning of facts about whole programs for the benefit of configuration management and building; and modifiability of the language from within (and perhaps also from without), to allow for purpose-oriented variability, and low-threshold implementation of abstractions over platform and product-line specific idioms.

I present a collection of technologies aimed at implementing such programming languages, and show a number of ways to apply such languages in ways that suit the niche platform application product line scenario. I use smartphone operating systems as an example platform ecosystem, and focus on error handling and prevention as an example concern that poses reuse, integration, and configuration management challenges in multi-platform codebases.

Contents

CHAPTER 1

Introduction

In this dissertation, I present a collection of technologies and describe an overall strategy for creating and maintaining a programming-language-centric tool environment for the production of cross-platform software. My specific aim is for the production tool setup to support the development of software applications built from a single codebase, but running across different smartphones. Niche platforms should furthermore be treated no differently than more mainstream ones, to hopefully lower the threshold of having a software product support less popular target platforms.

By *platform* I mean a base on top of which a software program can run, be it an operating system (OS), a virtual machine (VM), a cross-platform application framework, or a hardware-embedded computing platform (such as Nvidia's CUDA parallel computing platform). By *niche platform* I mean any platform that is not both popular among developers and backed by a large ecosystem of software and tool vendors. A platform may be niche by design, due to having been designed for a very specific purpose. A general-purpose platform may remain niche for lack of mainstream appeal, perhaps due to heavy competition or insufficient marketing. It can also happen that a platform with a large market share is nonetheless unappealing to developers, for example due to a poor developer experience and lack of incentives [Wood, 2014].

Like any other platforms, niche platforms have their challenges when it comes to a developer targeting them for the first time. There are costs be borne: some platform specifics must be learned, and probably some compatible code needs to acquired, ported or written from scratch. For that process, compared to a popular platform, there probably is less support to be had from peers, and fewer existing tools and libraries to be found. Some of the relevant details about a platform (and its tools) may be poorly documented, if at all; defects and quirks especially may lack documentation, and they may be more common for platforms that are less well exercised.

I do not believe there is a simple remedy for these problems, but I do believe that the problems can be contained in at least two ways. Firstly, one can choose to develop families of similar applications (e.g., only PIM applications, or only social networking applications); then, even if one must implement components from scratch, separately for each platform, there is likely to be

extensive reuse across each application family. Secondly, one can develop tooling for the management of such product families in a cross-platform setting. Such tooling should be reusable and adaptable, at least for adapting to different target platforms, and perhaps even to product family specific needs; as niche platforms may come and go, adaptability seems preferable to building in support for a fixed set of target platforms.

In the context of product families, I use a variety of terminology. A *component* is a composable piece of software that implements a known interface. A *core asset* (or just *asset*) is a component, or any other digital artifact out of which individual products of a family might be built. A *product line* is a product family whose individual members are produced systematically, by building them out of a common pool of assets, with the help of methods and tools. A *product-line architecture* (PLA) is a way of organizing assets and applying methods and tools for maintaining product lines, and a *production tool* is a tool that is used for that purpose. *Domain engineering* or *product-line engineering* (PLE) means the creation, maintenance, and organization of reusable assets of a domain, which might be that of a product line.

This dissertation's technological contributions aim to facilitate the creation of production tooling. When working on those contributions, I made some assumptions about what kind of technology might help achieve fit-for-purpose results; more specifically, I assumed that

> A family of special-purpose-adapted programming languages built on common platform-agnostic infrastructure and translating into platform vendor supplied languages can serve as a basis for mechanizing various cross-niche-smartphone software product line engineering tasks.

I arrived at that base assumption firstly through Lisp influences[1], and the Lisp tradition of customizing languages and integrating functionality into them, and secondly based on the idea that if we are working with a family of products with some variation between its members to suit different purposes and targets, might we not also have a family of languages in which to program them, similarly with variation to suit different purposes and targets. Having multiple languages allows for different design tradeoffs to be made, depending on the intended purpose.

For lack of more established terms, I use the term *domain-oriented language* for a mostly-general-purpose programming language that has (or can assume) *some* specialized features or characteristics to make it a better fit for its intended domain or purpose. I use the term *domain-oriented programming* to refer to programming with a collection of such languages sharing a similar "look and feel."

For economical implementation of such families of similar languages, it is probably useful if domain specializations themselves can be expressed as modular assets, perhaps in terms of composition-friendly formalisms such as attribute grammars [Knuth, 1968] or "funcons" [Churchill et al., 2015; van Binsbergen et al., 2016]. To satisfy this dissertation's modular language defi-

[1] Due to my use of Racket (a descendant of Scheme and Lisp) as a "language construction kit" of choice, Lisp influences are likely to show extensively both in the solutions and the vocabulary of this dissertation, in a way making it concern both niche platforms *and* niche languages at the same time, for which I apologize.

nition needs, I have opted to rely on the proven syntax definition mechanism of hygienic macros [Clinger and Rees, 1991; Kohlbecker et al., 1986], designed for safe composition[2], and also for extending languages from within. More specifically, I use the *Racket* programming language [Flatt and PLT, 2010] and its macro system and other machinery for defining languages as libraries.

Macro systems are suitable for various kinds of syntactic language adaptation, and the immediacy of writing a macro should also encourage on-demand language adaptation by product programmers as requirements change or new design patterns or better abstractions are discovered. Macros are not, however, a complete solution for engineering domain-oriented language implementations: macros transform a language into its "core" syntax, but my assumption is that multiple different core languages may be required, and that further translation to various different vendor languages is required.

This dissertation presents compiler front and "middle" end implementation techniques that aim to promote language infrastructure reuse in the context of such requirements. The presented techniques cover macro, module, and build system integration, macro-based preparation for compilation to other languages, and abstraction over program representations used in performing transformations.

This dissertation also suggests uses of language features towards automating aspects of product-line programming; for example: "alerts" for cross-platform error handling, a static component system for product assembly, and API access permission inference as part of configuration management. We presently do not have a single programming language which is both extensible and has all of those features; rather, we use a mixture of our research languages Erda, Magnolia, and Magnolisp for purposes of illustration.

This dissertation furthermore outlines a production tooling strategy that is centered around a "mouldable" programming language (of a combined nature of those research languages), with the idea being that the language will, for example: incorporate production support functionality; mould itself to meet the needs of different product families; integrate with both generic and platform-specific tools and languages; and be supported by its own associated, language-aware programming environments.

Ideally, I would like to show that the presented strategy can serve as the basis of a *comprehensive* set of product-line tools, such that it makes niche platform support *cost effective*, for many different kinds of product families. Alas, this has not been established, empirically or otherwise, although previous findings by Voelter [2014] from applying similar tooling to a similar domain (i.e., embedded systems development) do support the assumption of achievable comprehensiveness and cost effectiveness.

1.1 Motivation

Since cross-platform development has its essential complexities, which are in some ways aggravated when niche platforms are involved, one ought to be

[2]Hygiene is the moral equivalent of lexical scoping at the macro level [Adams, 2015]. Hygienic macro expansion ensures that names retain their original sites' lexical scope determined meanings, which helps avoid unintended interactions between different macro definitions and uses, for more reliable composition of program fragments.

sufficiently motivated before attempting to architect a cross-niche-platform codebase.

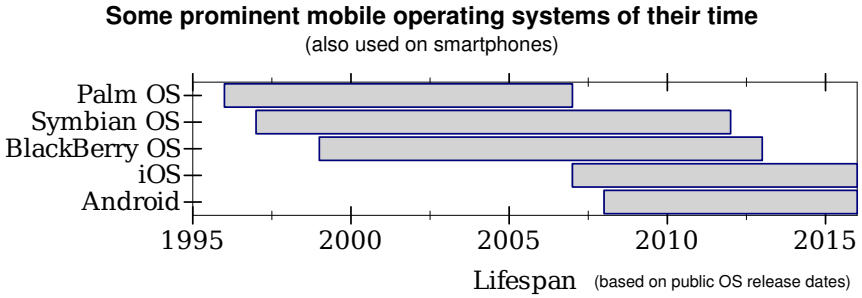
1.1.1 Why Care about Niche Platforms?

Some might question the usefulness of being concerned about better arming developers to target niche platforms. Is it not easier to just have the entire user base switch over to mainstream platforms?

Switching to better-supported platforms can indeed be a pragmatic decision where suitable platforms are available, and one has control over platform choices. An embedded application, for example, might come with bundled, dedicated hardware, without exposing the underlying platform to the end user, making the choice of that platform a mere implementation detail. When it comes to user-facing platforms, one might be in a position to influence (or dictate) the platform acquisition policies of an organization (e.g., which smartphone models are "standard issue"). Switching has its costs, however, and non-IT businesses in particular can be hesitant when it comes to adopting the latest operating systems.

There also are niche application domains for which there are no suitable mainstream platforms to which to switch. For example, the highly parallel hardware setting of a modern GPU does not host a general-purpose system of today, and one must choose to program against specialized offerings such as CUDA or OpenCL. Even when the hardware is capable of hosting a mainstream system, an application may require platform characteristics that no mainstream platform provides; for example, some of the most popular Linux distributions (e.g., Debian and Ubuntu) are also widely deployed at the higher end of the embedded hardware spectrum [UBM Electronics, 2012], but do not provide the real time and robustness guarantees required by some embedded applications.

Even in cases where switching platforms is a viable option, there may be reasons to keep supporting a niche platform. Android and iOS have overtaken the smartphone market for the time being, but for instance the previously dominant Symbian OS still has an installed base, due to having shipped in large volumes for many years (particularly on Nokia smartphones).



A software developer of today might still find some attraction in Symbian: it is largely abandoned, and thus no longer a "moving target;" its quirks are by now well known in the right circles; and there is still a dearth of off-the-shelf applications for it. An educated end user might choose Symbian for low price, recycling opportunities, or a wealth of choice of presently unfashionable physical device form factors (e.g., "candy bar," "slider," or "flip" phones, or

Figure 1.1: Some of Nokia's Symbian-based smartphones (from top-to-bottom and left-to-right): the candy-bar-shaped Nokia 6120 Classic; the taco-shaped, gaming-oriented N-Gage; the horizontal-touchscreen-enabled Nokia 7710; the square-shaped Nokia 7600; and the QWERTY-equipped Nokia E71.

designs with a hardware QWERTY keyboard or a swiveling screen); some example form factors are shown in figure 1.1.

1.1.2 Challenges in Niche Platform Development

Writing code specific to a platform with little mind share is a risky investment as the platform may get discontinued with little notice.[3] When writing a new application for a platform of uncertain future one likely wants to keep the codebase somewhat portable to other platforms. Writing portable code for heterogeneous platforms is difficult, however; even when their developer offerings support *some* common technologies (e.g., the C++ language), the commonalities are unlikely to cover everything that is required in a full application.

Rapid improvements in the hardware specifications of a particular device category also make portability harder. Many of the design decisions made initially for the Symbian platform, for example, were due to hardware constraints no longer considered current by the time smartphones overtook the "feature phone" market. Platform-level design decisions tend to some extent be exposed to application programmers as well, and have to be accounted for in writing portable software. Fortunately, platform vendors typically try to hide such decisions in any provided cross-platform APIs, such as any POSIX compatibility layer.

In the case of niche platforms, one tends to be highly dependent on vendor-provided developer offerings, which in turn tend to be quite different between

[3]Sometimes, as in the case of Nokia's Symbian-based S60 platform, the manufacturer announces a schedule for discontinuation ahead of time, but this appears to be harmful to sales [Wood, 2014].

vendors, and of varying quality. In a cross-platform scenario one then has the challenge of learning to use each platform's tools effectively, or—as I advocate in this dissertation—setting up a common, familiar set of general-purpose tools as a way of reducing exposure to platform-specific tools. When setting up such general-purpose tools, and adapting them to niche platform specifics as required, one should be prepared to go it alone; niche platforms have small developer communities, after all, consisting of individuals of different tastes and skill sets when it comes to tools development.

1.1.3 Cost Effectiveness of Targeting Niche Platforms

In considering the potential rewards of targeting niche platforms, I make the general assumption that the smaller size of a niche platform's potential customer base is offset by better discoverability of applications. This is a reasonable assumption for small vendors without large marketing budgets; if one cannot afford to "buy" visibility in a sales channel, it helps discoverability if there are few available applications of similar functionality, due to few developers targeting the platform. Granted, popular platforms have far more sales *potential*, but a realistic software vendor acknowledges that best-seller applications realizing that potential are an exception rather than the norm. Also, as competition tends to drive prices down, a niche platform may allow for higher per-unit pricing; thus, for as long as the higher unit sales potential of popular platforms is not realized, a less popular platform may well be *more* profitable.

If the expected rewards for an average application are comparable, then cost-effective development for niche platforms boils down to achieving comparable development costs, despite any lack of supporting facilities in the platform ecosystem. Language is a tool for abstraction, and my thesis is that it is possible to an extent to domain engineer an application codebase in which each target platform is just a variation, to be abstracted over in terms of common language technology. The idea is that at least at the level of configuration management and software composition each platform should be treated alike. Any vendor-specific tools and libraries used for other aspects of development can be of varying quality, which is where differences in cost arise.

1.2 Domain Engineering

There are many aspects to domain engineering [Harsu, 2002], but in essence, it is a discipline for developing software for reuse. As in any *engineering* discipline, the aim is to be systematic about the way one goes about doing that, whether it comes to methods, tools, or practices.

In the context of cross-niche-platform development, I see domain engineering as a possible way to tame complexity, by systematically capturing platform knowledge as code and other reusable artifacts. Real-world applications can easily get quite complex (accidentally or essentially), and multi-platform, multi-product lines have additional dimensions of complexity. One must learn the specifics of the target platforms (e.g., discover their defects and find workarounds), and find solutions to meet the specific requirements of the product line. Through domain engineering, we can hope to capture and reuse

that information as artifacts, be they components, language extensions, `make` rules, or something else.

For purposes of domain engineering, then, we might hope to write code in a language that is able to directly express manageable-size units of reuse that are suitable for composition, rather than having to rely on design patterns to encode them in terms of other constructs (e.g., a mixin can be represented as a C++ class with a parameterized superclass). In other words, we want to be able to *identify* some entities out of which software might be composed, and *encapsulate* them as first-class entities in the language, so that we can *integrate* them in different compositions with the help of a composition mechanism provided by the language [Sunkle et al., 2008].

In this dissertation, I focus on "components" consisting of data structures and associated operations as such language entities, and "component systems" as expression and composition mechanisms for those entities; I elaborate on the nature of those entities and systems in section 1.4.2. It would be possible to do domain engineering in terms a different kind of unit of composition. For example, a *feature* is an increment in program functionality, and Prehofer [1997] has suggested having a **`feature`** construct in a programming language.

In our context it is not merely the choice of unit of reuse that is interesting, but also the interchangeability of such units. Since the introduction of feature-oriented programming, for example, there has been debate as to whether it is important for features to have interfaces [Kästner et al., 2011]. For organizing reusable cross-platform code—regardless of the chosen unit of reuse—it would appear particularly useful for the units to have (abstract) interfaces [Britton et al., 1981], which can be detached from specific units. This is because it is useful to be able to choose from alternative implementations of the same functionality, to pick one that is suitable for the target platform, without affecting code that uses that functionality.

Assuming suitable language support for encapsulation (with interfaces), I believe that domain engineering can be a practical solution for abstracting over platform-specific ways to access platform services. It is normally APIs that provide access to system services, after all, rather than some functionality built into a language. To abstract over the required services, one can define corresponding platform-agnostic interfaces, and implement them for all the targeted platforms. I do not believe it is feasible to maintain implementations of complete, correct, and current platform abstraction layers for multiple evolving platforms, and even if that were feasible, it would hardly be cost effective to do so for niche platforms. Luckily, a single product line is unlikely to require all conceivable system functionality, and it may be feasible to maintain large enough system abstraction APIs to cater for product line needs, even if those APIs must be maintained in-house by an independent software vendor (ISV).

1.3 Configuration Management

In this dissertation, I consider the term *configuration management* to mean systematically dealing with different software system configurations. Methods, tools, and processes for doing that become particularly important in the context of product-line engineering, where the goal is to be able to easily scale to

a large number of different product configurations; indeed, some have found that switching to a product-line practice can quickly enable the enlargement of a product portfolio [Hetrick et al., 2006].

A central aspect of configuration management is to identify the different configurations, and to keep track (or deduce) facts about their properties, to support repeatable builds, for example. The term "configuration management" has for decades been used in a somewhat broader sense to mean a systems engineering process that is not only concerned with configuration identification, but also the tracking of configuration changes through product lifetimes. A more narrow definition suffices here, as change control is not discussed. However, it is implicitly assumed that software assets are stored in such a way that changes to them can be tracked with a version control system (e.g., Git or Subversion). It is furthermore assumed that the assets may include artifacts stating facts about different product configurations, which then can also be version controlled.

I use the term *variant* to refer to any valid configuration for a system, whether its codebase has been domain engineered or not. I assume that there is some specification formalism that may be used to specify configurations, and that a large enough set of *configuration parameters* (specified in terms of that formalism) uniquely identifies a variant.

Modeling of variants as sets of configuration parameters is related to the concept of "feature modeling." That concept was first introduced as part of *Feature-Oriented Domain Analysis* (FODA) [Kang et al., 1990], which is a method for discovering and representing commonalities among related software systems. A *feature model* [Batory, 2005] defines features and their usage constraints in a product line, and thus provides a closed-world view of all legal feature combinations. Due to our cross-platform development focus, we should perhaps be more concerned with different *implementations* of functionality than on features in distinguishing product-line members. After all, a cross-platform application codebase is likely to contain platform-specific implementations of individual components, and the application must be composed out of target-compatible implementations.

We might get quite far by expressing such target-specific compositions in terms of a suitable component system's mechanisms, but that alone is not sufficient if we want our configuration management solution to make it possible to capture further domain knowledge about target platform variability. Such knowledge is essential for building; even the exact same software composition (using the same APIs) might link against different external libraries, and have to be built with different toolchains and options depending on the target. There are also likely to be differences in the way the software has to be packaged for deployment, and that process might also involve code signing (and key management). In our case we also have to make sure that we pick the appropriate target language to generate for any components that require translation into vendor-toolchain-supported languages.

It is safe to say that configuration management challenges compound in a cross-platform setting due to target platform variability. There can be significant differences not only between platforms, but also platform revisions. The same can be said about different releases of platform-specific software development kits (SDKs). Tizen's native programming offering, for example, has seen significant changes over time: a native application framework was in-

troduced in Tizen 2.0, with its C++-based APIs seemingly derived from those of the *bada* [Morris, 2010] operating system; Tizen 2.3 replaced the framework with a C-based one that is accompanied by the Enlightenment Foundation Libraries (EFL).

From a build configuration point of view, the most significant differences would tend to be platform API changes and build toolchain changes. When building software, both the intended target platform *and* the SDK used for building tend to matter; in particular, when mixing and matching target platforms and SDKs, one should be sure to use only common subsets of target platform APIs, lest either build or execution fail due to missing APIs. There can be incompatibilities beyond APIs; for example, Symbian's EKA2 real-time kernel introduced support for DLL global writable static data, but many SDKs for EKA2-based platform releases shipped with a compiler that was defective with respect to that feature [Hasu, 2010].

To address the challenges of maintaining information about interesting legal compositions of product-line assets, we might seek to minimize the number of details that we need to maintain manually. One way to do that is to derive some configuration parameter values from others. Another way to do that is to infer some properties of component compositions by analyzing the code of those compositions; suitably designed programming languages can both mechanize the creation of those compositions, and allow for more accurate static analysis. A third way is to exploit any useful domain knowledge in vendor-specific build tools; for example, it may be that by providing a small number of details about a piece of software being built, a vendor build tool can deduce a suitable way to invoke compilers and other auxiliary build tools.

If the aim is to support even lone software developers in flexibly coping with their cross-platform product portfolios, I believe it is most practical to have lightweight tools without rigid requirements for completeness and correctness. Therefore, rather than doing closed-world variability (or feature) modeling, configurations are perhaps better modeled as partial specifications in an open-world setting. More specifically, any constraints should not have to be complete, and it should be sufficient to deal with one variant at a time.[4] The tools should furthermore assume little about target platform offerings, due to their varied nature; incorrect assumptions might hamper the use of any vendor tools, and the domain knowledge encoded in them.

1.3.1 The Konffaa Configuration Manager

I have previously presented a lightweight command-line-based configuration manager [Hasu, 2010, section 7.2], whose more recent incarnation—named *Konffaa*[5]—was used in managing the different configurations of the Anyxporter application, discussed in this dissertation's section 4.5. Konffaa is an example of a tool for maintaining sets of variant-specific configuration parameters, some of which can be computed based on explicitly specified ones. Especially parameters specifying the target platform's name, version, and SDK version

[4] At any given time, a domain engineer may have some products under development or temporarily or permanently unmaintained, and it would be unhelpful to require that all assets and configurations of a product line be kept complete and consistent at all times to avoid complaints from management tools.

[5] Documentation: `https://bldl.ii.uib.no/software/pltnp/konffaa.html`

can often be used to deduce many others, typically relating to available tools and system APIs. As an example of platform knowledge captured with a configuration manager, a parameter indicating the availability of the Music Player Remote Control API for the S60 platform might be defined for Konffaa as[6]

```
(define-attribute have-mplayerremotecontrol
  (and (<= 31 s60-vernum 32)
       (= kit-vernum 31)))
```

Konffaa's variant specification language is implemented as a `#lang` for Racket (the `#lang` mechanism is explained in section 2.3.1). The *`konffaa`* language augments Racket with syntax for specifying variants. The syntax is macro-defined sugar on top of a purpose-built object system, whose mechanism for multiple inheritance may be used in expressing commonalities between variants, by inheriting named member values and constraints. The object system distinguishes between public (exported and serialized) "attribute" and private (unexported) "field" value members, whose values are computed lazily and memoized. Similarly to Magnolia's support for declaring semantic-constraint-specifying axioms, Konffaa also has syntax for specifying constraints between fields as "axioms," to be used for checking the validity of a chosen variant. For example, all S60 configurations might inherit a check for S60 platform and SDK binary format compatibility:

```
(define-axiom s60-kit-binary-compat
  (assert (or (and (< s60-vernum 30) (< kit-vernum 30))
              (and (>= s60-vernum 30) (>= kit-vernum 30)))))
```

Konffaa operates by processing an input file that describes a variant, and by then computing a full set of attributes for that variant, which—if the associated axioms hold—are then output essentially as sets of key-value pairs in a variety of different file formats. The output files may then used to configure other development tools (most notably build managers) in a variant-specific way.

General-purpose build managers found on niche smartphone platform SDKs include GNU Make and Ninja; more special-purpose ones include the Qt cross-platform application framework's `qmake`, Tizen's `tizen`, and Symbian's ABLD and SBSv2. Konffaa presently supports GNU Make and `qmake` as output languages, as well as C and Ruby, with the latter two aimed at configuring programs and custom build scripts. Scripts can be particularly useful for driving template-based generation of input files for platform-specific build tools, with which the tools can be invoked as normal in order to benefit from their platform awareness.

1.3.2 Program Build Configurations

Configuration-manager-maintained, variant-describing configuration parameters are abstract requirements (e.g., which features are required and what

[6]Konffaa is parsed like the Racket language. Prefix notation is used, so that the first symbol within a pair of parentheses names that form, whose meaning is generally given by the binding of that name. By convention, Racket and other Lisps use the "minus" sign (rather than a hyphen) to separate words in names.

platform is targeted) for building a product implementation out of components and any other assets. As there may be any number of ways to assemble a product that meets a given set of requirements[7], the requirements should be concrete enough that *in combination with* any knowledge and decision-making logic encoded in subsequently invoked tools, the build process will consistently arrive at a specific composition and build configuration for the product.

As a baseline case, each maintained variant configuration might have a parameter that simply names the desired program composition satisfying that variant's abstract requirements, with the name being that of a source file expressing the composition in a programming language. Another parameter might name a script for building that composition into an executable, with any packaging required for deployment. Both the program and the build script could be somewhat generic, and parameterized with the configuration; conditional compilation is commonly used in C and C++, for example, as are variables and conditionals in GNU Make.

Even if we choose to use a composition language that has special-purpose language for component composition, allowing concise and explicit expression of desired program instances, that may still not fully realize scalable code reuse across a product line. If our language targets a specific C++ build toolchain, then our program must have the appropriate **`#include`** directives, and we must build all the C++ source files required by that program composition, with suitable build options, linking the executable against the required libraries, etc. If we do nothing to mechanize the determination of suitable build configurations, then our ability to scale to large numbers of product variants is likely to be hampered.

As observed by de Jonge [2005], reuse between software systems is often suboptimal due to modularization principles only being applied to structuring program functionality, while neglecting the possibility of applying the same principles to the build level. This culture of neglect may in part be due to limitations of traditional *Make* [Feldman, 1979] implementations, which make completeness of build dependencies only achievable in a single makefile, and thus discourage attempts at componentization of build information.

For a product line, at least, it would seem worthwhile to overcome any limitations of build managers in order to represent build dependency information in a modular and reusable way. There are different approaches we might attempt. If we simply adopted a separate build-level component system for expressing build dependencies, perhaps in terms of a build manager that supports components [Dolstra, 2003], there would still be a disconnect between program components and build components; given a program composition, we would have to somehow determine which build-level components (e.g., object files) it requires.

To avoid that disconnect we might instead use a component system that combines program and build components; Knit [Reid et al., 2000], for example, is a component system for C such that its **`unit`**s can express not only required and provided symbols, but also **`files`** to build and compiler **`flags`** with which to build them.

[7] Even for a single component interface there might be multiple interchangeable implementations from which to choose; for example, the asset pool might contain both GnuTLS and OpenSSL based implementations of certain encryption-related operations, and a given variant's target platform might have both of those two popular libraries.

Knit has the right idea in that expressing compositions in such a language makes it straightforward to arrange for that language's compiler to deduce the overall build dependency information for a given composition. However, for a cross-platform (and possible even cross-language) setting it is too specific; for a system dependency, we should not have to name specific files or flags, as even the same library can have a different file name on different platforms, or its linking may be different (e.g., object file, static library, or a dynamic library), or the required compiler options or include paths may differ.

We could achieve "late binding" [de Jonge, 2005] of such build specifics through indirection, for instance by binding build information to abstract names, leaving it to other tools concerned with platform specifics to determine the semantics of those names for a chosen target.

Having such late-bound names is not unlike the way GNU Autotools' `autoconf` allows the definition of dependency parameters, which may then be passed to the `configure` scripts it generates, with the syntax `--with-`*`name`*`=`*`value`*. A source-level component system (such as the one in Magnolia) might allow each component implementation to explicitly state its dependencies by listing such dependency names (e.g., `sqlite` for a component that uses the SQLite database API), to be resolved into concrete dependencies later as necessary. For finer than per-component granularity, we might even annotate individual types and operations with their build dependencies. Magnolisp lacks a component system, and infers `build` facts at an operation-level granularity; its `build` annotations can list dependency information symbolically, or as filenames:

```
[build sqlite (+= headers "f.hpp") (+= sources "f.cpp")]
```

An example case of whole-program inference of facts annotated for operation implementations is the permission inference solution described in chapter 4. In a typical smartphone security model, missing permissions trigger run-time errors, meaning that it is sufficient to request permissions only for operations that might get used; build dependencies, on the other hand, must typically be requested for any operations whose invocations appear anywhere in a program, to avoid build or link time errors. The Magnolisp compiler removes that distinction by optimizing whole programs to eliminate unreachable operation invocations and implementations, which also helps avoid unnecessary `build` dependencies in inference results.

1.3.3 Deployment-Time Components

So far I have discussed managing program compositions and build dependencies in terms of components, but components can exist not only during software development and building, but also during deployment. The *Maak* [Dolstra, 2003] build manager supports components, and goes as far as unifying build and deployment by turning building into deployment, and by consequently also having knowledge of the relationships between binary components. In a niche platform context, however, we cannot generally expect to be able to adopt such a solution pervasively, making it of limited use in managing binary dependencies. Smartphones, for example, have various restrictions for installing and running code natively, and the common assumption of cross-compilation is another complication for build and deployment unification.

Maak, by its nature, acts both as a build manager and as a package manager, but not all platforms have or require the latter. Platforms that to some extent isolate applications (and their code) from each other, for instance, probably have less need for a package manager capable of expressing dependencies between installation packages (containing binary components and other run-time assets). For purposes of this dissertation, I assume that each product is deployed as a single package, and that any run-time dependencies beyond that package are a part of the platform. If those dependencies must be declared to ensure availability (or similarly, if associated API access permissions must be declared), then they translate to *build*-time requirements for correctly declaring them.

1.4 Mouldable Programming Languages

> "The ability to reason abstractly, to see generality through the particular, and then to particularize the general, are very useful for the development of high quality software."
>
> Kapur, Musser, and Stepanov [1981]

Programming languages often require many years of development before fully realizing the potential of their design principles and goals. So it is with the programming languages that we have developed, and which I use for illustrative purposes in this dissertation. Magnolia and Magnolisp in particular are research prototypes of languages that aim to be "mouldable"[8], but as yet neither one of them individually embodies all the facets of *mouldability*, i.e., flexibility, adaptability, genericity, and robustness; taken together, however, they do.

Magnolisp flexibly integrates with other tools, and is adaptable through its adopted `#lang`uage definition mechanism. Magnolia's component system facilitates generic-but-specializable definitions, with axioms supporting algebraic specification of semantics for components so that mechanized reasoning about compositions is possible (consequently, one can hope for more robust compositions through checking and testing of applicable specified semantics). These features of the two languages are not mutually exclusive by their nature, and hence I believe that they can be integrated into a single language; we are not there yet, but chapter 6 envisions how cross-niche-platform development in such a "fully mouldable" language might look. In this section I discuss our existing languages and their niche-platform-friendly features and characteristics.

1.4.1 Magnolia, Magnolisp, and Erda

It may not be terribly hard to port basic run-time support for a feature-rich programming language (e.g., Racket) for a given target platform, but it might also not be all that useful to do so; for writing non-trivial applications mere language constructs are not enough, as access to system services is required. I have

[8] `http://ii.uib.no/mouldable`

discussed configuration and build managers, but a cross-platform domain-engineering setting would additionally appear to call for an "API manager."

Magnolia [Bagge, 2009] is a language for API management and reuse. Its algorithmic language, while unusually constrained in order to facilitate static analysis, is otherwise quite ordinary in that its constructs are few and mostly familiar from other languages. Its component language, on the other hand, is exceptionally capable, and designed to promote conceptual use of APIs, and generic and compositional implementation of APIs.

Magnolia features a static component system that supports *external linkage*, meaning that a component may refer to others indirectly through a parameterization mechanism [Culpepper et al., 2005]; in that respect its component system is like those of nesC or mbeddr, for example. Magnolia goes beyond merely syntactic component interfaces, however, and follows in the footsteps of Tecton [Kapur et al., 1981] in integrating programming and specification at the language level, in order to support safer composition. Specifications may be incorporated as *axioms* stating semantic constraints as universally quantified logical expressions relating operations to each other [Bagge and Haveraaen, 2014]. Axioms, in turn, may be defined as part of *concepts* [Gregor et al., 2006], or interfaces with integrated specifications of expectations about the behavior of their operations.

Magnolia's emphasis on APIs shows also in the way operations can be invoked uniformly regardless of whether they were declared as **`function`**s or **`procedure`**s, through a process of "functionalization" to derive functions from procedures, or "mutification" to translate function uses into procedure calls [Bagge and Haveraaen, 2010].

Magnolia compiles to other languages, with C++ presently being the usual choice, one that is also supported by many niche platforms. The compilation machinery is Eclipse IDE integrated [Bagge, 2013], and the constrained algorithmic language means that there is ample potential for the implementation of sophisticated refactorings and other interactive, assistive features. Magnolia is further introduced in section 4.3.

Magnolisp is a language that I conceived to experiment with the areas in which I felt Magnolia was still lacking, namely integration with other tools (or use standalone), large-scale core asset management, and syntactic "self-extensibility" [Erdweg et al., 2012]. Superficially, Magnolisp resembles Racket; it was not a goal to innovate in the area of algorithmic language syntax, and familiar is better for remembering. Unlike Racket, however, Magnolisp is designed for ease of static reasoning, by making language-semantic choices similar to Magnolia's. Magnolisp also has a core that is designed for easy deployment, in that the core is straightforwardly translatable into other languages.

Magnolisp is not tied to an IDE, and its compiler is quick to start up, and has both a command-line interface (CLI) and an API. Similarly to Konffaa, the compiler can also output build information in a variety of languages, which helps with flexible use and integration. Magnolisp does not "externalize resource management" [Felleisen et al., 2015] by having IDE-managed "projects," but rather it is enough to express a program configuration in the language; there are no complaints about modules that are not loaded for that program, which enables a "divide and conquer" approach to large domain-engineered codebases. Magnolisp is also an experiment in macro language and system reuse,

as it integrates with Racket to reuse its facilities for language self-extension. Magnolisp is discussed in more detail in section 2.2.

Magnolisp's exploitation of Racket's language definition machinery means that it is possible to create *families* of languages on top of the Magnolisp infrastructure. For example, while `magnolisp` is Magnolisp proper, there is also a `magnolisp/base` language for implementing the Magnolisp run-time library, and `magnolisp/2014` as a language for backward compatibility. Similarly, our Erda$_{C++}$ language with experimental failure processing syntax and semantics is implemented as the Magnolisp-based language `erda/cxx`. While our more established Magnolia language already has **`guard`** syntax for declaring partiality of operations in an abstract way, and **`alert`** syntax for declaring possible concrete error conditions, the language is still lacking in the area of error propagation and handling; the Erda family of languages (which includes Erda$_{C++}$) is our tool for exploring such error management facilities without disrupting the development of Magnolia. Erda is discussed in more detail in section 5.4.

1.4.2 Module and Component Systems

In the context of domain engineering and large-scale systems development, the modularity mechanisms of the used programming languages become important. While many languages are lacking in this area, they do tend to have some mechanisms for splitting a system into smaller sub-systems of an internal implementation and an external interface [Kästner et al., 2012]. There are two subcategories of such modularity mechanisms that are particularly relevant to this dissertation, and I use the (somewhat overloaded) terms "module system" and "component system" for them.

A *module system* is a compile-time code organization facility, without external configuration or separate interfaces. In such a system code is organized as *modules* that have their own namespace, and fixed static imports and exports.[9] For example, Magnolisp adopts Racket's module system [Flatt, 2002] for its code organization, a system which is suitable for compile-time composition and namespace management and the like, more recently even supporting nested modules [Flatt, 2013]. It is not ideal for expressing compile-time variability, however, as it lacks a parameterization mechanism to cater for variability inside a module. It is also not ideal for abstraction in cross-platform development, as module interfaces are not separate from implementations.

A *component system* is both a code modularity and reuse mechanism, with separate interfaces to allow for programming against abstract APIs, and with configurable implementations to allow for specifying internally used APIs' implementations. In such a system code is organized as components that have at least an (exported) *interface*, sometimes called its *provides interface* [van Ommering et al., 2000]. A component may be fully or partially implemented (or not at all), and any missing parts of the implementation must be specified as parameterizable imports, which constitute the component's *requires interface*. Such interfaces make it possible to obey the "principle of external connections," which is to define components separately from their connections [Flatt, 1999].

[9] In Magnolia terminology any top-level definition is a "module," but I avoid using the term in that sense here.

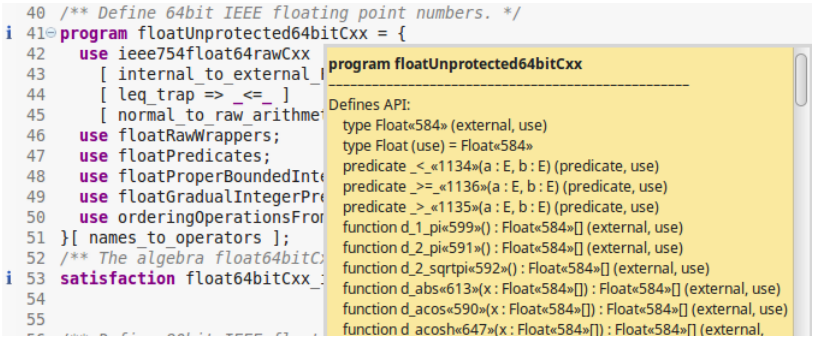


Figure 1.2: Hover documentation in Magnolia's IDE.

Depending on the component system, composition may happen statically (at compile time) or dynamically (at run time). Racket's "units" system [Flatt, 1999] is an example of a hybrid system whose components are first-class values, but in which static information about components is exploitable for convenience in specifying compositions. Examples of static-only component systems fitting my definition are those of Knit [Reid et al., 2000], Koala [van Ommering et al., 2000], and nesC, which are all languages geared towards embedded software development.

A specific motivation for having a static-only system in Magnolia is to enable components to serve a role similar to higher-order functions in supporting parameterization, while still leaving all operations resolvable at compile time. For example, upon seeing the code for the higher-order Racket function[10]

```
(define (modify-first! vec modify)
  (vector-set! vec 0 (modify (vector-ref vec 0))))
```

we cannot inspect the `modify`ing operation for purposes of reasoning. Magnolia has no higher-order functions, but the `modify` operation could instead be specified as part of a component's requires interface.

Magnolia is unusual in its support for concepts, i.e., interfaces defining both syntax and behavior.[11] Another way in which Magnolia differs from Knit, Koala, etc. is that component implementations' exports need not be listed explicitly, but rather everything that is implemented (or **use**d from other implementations) gets exported by default; this design choice easily results in large export sets, making tool-provided information (e.g., hover help in an IDE, as shown in figure 1.2) quite important in understanding Magnolia code [Bagge, 2013].

Magnolia's component interfaces (as declared with **concept**) may list **type**s, **function**s, **procedure**s, and **predicate**s as named abstractions over data

[10] Racket's ***define*** form binds a name to a value, with the name in this case being `modify-first!`, and the value being a function. Instead of using **define**'s function definition shorthand form, an equivalent definition could be written as `(define modify-first! (λ (vec modify) ....))`.

[11] Magnolia's component language does include a **signature** operator for stripping out the axioms of a concept, in order to get a "plain" syntactic component interface.

structures, expressions, statements, and predicate expressions. Any **implementation**s are given separately, and may be natively in Magnolia, or **external**ly (and opaquely to Magnolia) in a target language (such as C++). Any parts of an implementation that are left abstract are specified with **require**, and a fully concrete implementation may be declared as a **program**, to have its operations made available for invoking through the program executable's CLI. An implementation's **satisfaction** of a concept is stated by declaring that it **models** the concept. A concept's **axiom**s—which abstract over **assert**ions—are a property of the concept, and they are implemented for it in Magnolia.

1.4.3 Algebraic Specification and Reasoning

Reuse of APIs is important in a product-line setting, while having alternative but semantically interchangeable implementations of APIs is useful for abstraction in a cross-platform setting. A language implementation tends to complain about syntactic mismatches between API declarations, implementations, and uses, but in aspiring to correctness, it can be useful to also have language support for helping to ensure matching semantics through specification and automated reasoning.

Magnolia's algorithmic language has been suitably restricted for more effective static reasoning, by disallowing or carefully controlling side effects, aliasing, and dynamic dispatch, for example. The language includes **axiom**s and **assert**ions for informing Magnolia about semantic constraints between different (algebraic) expressions using the operations of a specific API. This kind of language-integrated algebraic specifications differ from the more commonly supported pre- and postcondition specifications, which relate a single operation's inputs to its outputs [Bagge and Haveraaen, 2014].

Magnolia's component language is quite agnostic with regard to the algorithmic language used, and should thus accommodate different selections of declarations, expressions, and statements [Bagge and Haveraaen, 2013]. Concepts do require *some* language for declaring abstract types and operations, of course, and there must also be language for defining implementations (i.e., data structures and algorithms) for them, and for invoking the operations; the algorithmic language of Magnolisp, for example, also meets these requirements.

Formally, a "many-sorted algebra" [Loeckx et al., 1996] can be used to capture the semantics of a Magnolia component implementation, thus providing a basis for reasoning about Magnolia's algorithmic language in the context of that component. Magnolia's component language, in turn, builds on the theory of "institutions" [Goguen and Burstall, 1992], which provides tools for reasoning about concepts and implementations and their satisfaction relationships; the language supports systematic changing of interfaces through "renamings," for example, making such reasoning non-trivial.

Magnolia's restricted language and integrated algebraic specifications open up many possibilities for static reasoning. If opaque target-language operations are suitably annotated, for example, inferring facts for larger program fragments from their code can be quite accurate, as discussed in chapter 4. Axioms, in turn, have applications to program transformation, formal verification, and testing, at least: axioms might be interpreted as rewrite rules in transforming programs, perhaps for purposes of API-specific optimization

by a compiler [Bagge and Haveraaen, 2009]; it may be possible to prove that a component satisfies its stated properties, for assurance of correctness; and axioms can serve as a basis for automated test generation.

1.4.4 Axiom-Based Testing

In *axiom-based testing* [Gannon et al., 1981], test cases are generated automatically for an API implementation based on applicable axioms, probably using randomly-selected, type-compatible data for the axiom arguments, with the programmer possibly providing some guidance about how to select the test data. In practice this might mean the language (or its implementation) providing a way to request a program whose "main" routine runs all of the program's axioms (containing assertion statements, as in Magnolia) under a test harness that collates and reports the results. One might create several different program configurations in order to exercise a large portion of a product line's components, thus avoiding much of the need to write unit tests by hand.

Ideally, frequently used test configurations would be such that the developer can run the tests on his or her workstation conveniently, without any interaction. In such configurations, one would avoid components that can only be tested on a niche target, or rely on workstation services or peripherals that require manual setup before use, for example. Where no suitable alternative component implementations are available, one can resort to "mocking" to create components used only in test configurations. A *mock* is an implementation of a component, object, or operation that imitates the behavior of a proper implementation.

Mocking can be particularly useful in testing embedded software, which is often developed alongside the hardware it is to run on; the target hardware may be undecided, unavailable or unfinished, or perhaps just too expensive to get for every developer [Grenning, 2011]. Niche platforms' APIs and services may suffer from a lack of attention, making them more likely to not behave as advertised; in discovering and narrowing down such problems, a cross-platform product line's selection of components to test with and platforms to test on can even turn out to be an advantage.

1.4.5 Source-to-Source Compilation

A programming language implemented by generating source code allows for reuse of existing infrastructure for the target language. I use the term *source-to-source compiler* (or *transcompiler* for short) for such language implementations. A transcompiled language can enable abstraction over cross-cutting concerns like target language versions, implementations (and their defects), and idioms. If the source-code-generating compiler furthermore produces human-readable output of a high abstraction level, then it also has a low adoption barrier in the sense that it can be regarded merely as tools assistance for programming in the target language. [Hasu, 2014]

The idea of generating source code for further processing by other tools is not restricted to the programming language proper, but can also encompass build utilities, resource compilers, etc.; e.g., Konffaa and Magnolisp can both generate build information in the GNU Make language (among others). Having the language export its knowledge about a program composition avoids

the need to manually maintain that program's build specifications, possibly in multiple formats (to support different build managers). The language implementation also has knowledge about implemented APIs, and one might also want to generate source code defining API bindings for other languages in addition to generating an implementation in the target language; e.g., Lua is a popular choice for application scripting, and Lua bindings can be implemented in C (or C++).

Like any compiler, a typical source-to-source one involves parsing and some intermediate (program) representations for analyses and transformations, and I discuss those implementation aspects in chapter 2 and chapter 3, with the former chapter covering not only parsing but also user-defined transformations (through macros) during parsing. Transcompilation also involves pretty printing to produce program text, possibly with code formatting for better human comprehension; we have discussed those compiler implementation aspects previously in a paper describing a code formatting architecture structured as a pipeline of "token processors" [Bagge and Hasu, 2013].

The Elements[12] toolchain by RemObjects is an example of shared infrastructure for transcompiled languages. As of version 8.3 it supports Oxygene, C#, and Swift as source languages, with *Oxygene* being the vendor's own design, influenced by Object Pascal. The choice of three essentially equivalent general-purpose source languages helps cater for different tastes, but a typical ISV would probably pick just one of them as the in-house standard. Elements supports C# (with .NET), Objective-C (with Cocoa), and Java as translation targets, thus enabling abstraction over those languages from a single source language and development environment.

Elements appears geared towards programming against target framework APIs, either directly, via "mapped types" (which map source types and their operations to target-specific ones at compile time), or via programmers' own (potentially cross-platform) APIs; in this respect its philosophy is aligned with my view that maintaining comprehensive platform abstraction layers (in the style of Qt) is cost ineffective in a niche setting. However, I would additionally contend that for any code that is in any case target specific due to its direct use of target-specific APIs, it might be useful to program in a target-specialized variant of the chosen source language, such that the language can be made to include select target language abstract syntax, and perhaps also further syntactic abstractions over target-idiomatic design patterns.

1.4.6 Macro Systems

A *macro* is a compile-time-evaluated function that maps syntax to syntax. *Macro expansion* translates away macro uses according to the macro-associated syntax mappings.

Lisp macros Lisp macros are *syntactic*, in that they operate on tree-like syntax representations rather than strings or token sequences.

Scheme macros Scheme is a flavor of Lisp in which macros default to being referentially transparent and hygienic, although it is still possible to explicitly

[12] `http://elementscompiler.com`

violate those properties as desired. A *referentially transparent macro* [Clinger and Rees, 1991] is such that its definition's uses of bound identifiers retain their meaning irrespective of the bindings in effect at the macro's use sites. A *hygienic macro* [Dybvig et al., 1992; Kohlbecker et al., 1986] avoids unintended capture of identifiers. The two main aspects of hygiene are to ensure that references introduced by a macro can only be captured by bindings introduced by the macro, and that bindings introduced by a macro can only capture references introduced by the macro [Adams, 2015]. Referential transparency and hygiene together cause macros to respect lexical scoping.

Schemers commonly write macros in a "macro-by-example" [Kohlbecker and Wand, 1987] sub-language with *patterns* for matching input syntax and *templates* for building output syntax; such a sub-language is defined by the standard **`syntax-rules`** [Kelsey et al., 1998] form, for example. What enables programmable-yet-hygienic macros in Scheme is its *syntax object* data type for lexical-information-enrichable S-expressions [Dybvig et al., 1992]. Even with program fragments reified as syntax objects, it is still possible to use forms such as **`syntax-case`** [Dybvig et al., 1992] to do pattern-based matching on syntax, with the added flexibility of being able to programmatically inspect and manipulate syntax as data.

Racket macros Racket is a descendant of Scheme, but its macro system has evolved further towards being a "wider compiler API" [Barzilay et al., 2011; Flatt et al., 2012; Tobin-Hochstadt et al., 2011]; I discuss its additional features and characteristics elsewhere in this dissertation as appropriate. Racket previously maintained hygiene during macro expansion by marking and renaming syntax objects [Flatt et al., 2012]. Flatt [2015] recently introduced a new model of macro expansion based on tracking bindings as sets of scopes, and this simpler model might provide a more approachable basis for introducing a similarly capable macro system into other languages (Magnolia, perhaps, in our case).

Racket macros: A primer As in Scheme, the syntax object data type plays a key role in Racket's macro system. We can augment a plain S-expression "datum" with lexical-context information to turn it into a syntax object, for example using the lexical context of another syntax object `ctx` to turn a plain symbol into an identifier:

```
> (datum->syntax ctx 'list)
#<syntax list>
```

It depends on `ctx` whether the resulting identifier refers to Racket's **`list`**-constructing function. Identifiers are further discussed in section 2.3.3.

Similarly to Lisps' traditional quasiquotation mechanism for building S-expressions, there is a quasiquotation mechanism for syntax objects: `#'` is a *syntax quote*, `#`` is a *syntax quasiquote*, and `#,` is a *syntax unquote*. For example, writing `#'(list 42)` is equivalent to writing `#`(list #,#'42)`. Syntax-quoted code is further discussed in section 2.3.5.

Racket's `#'` notation also supports *pattern variables* as an alternative (or complement) to using quasiquotation. For example, for another way of con-

structing the above syntax, we can bind the literal syntax `42` to the pattern variable `num` with

```
> (with-syntax ([num #'42])
    #'(list num))
#<syntax:4:0 (list 42)>
```

The ***with-syntax*** [Dybvig et al., 1992] form is similar to **let**, but introduces pattern variable bindings rather than program variable bindings. The term "pattern variable" is descriptive as **with-syntax** actually performs pattern matching between the left- and right-hand sides of its binding forms. Thus, for additional deconstructing, we could also write

```
> (with-syntax ([(num _) #'(42 (43 44))])
    #'(list num))
#<syntax:5:0 (list 42)>
```

Another similar syntax pattern matching form in Racket is the Scheme-standard ***syntax-case*** [Dybvig et al., 1992] form, which also binds pattern variables. The definition of the `static-cond` macro later in this section is one example of the use of **syntax-case**.

The ***syntax-rules*** [Kelsey et al., 1998] form is similar to **syntax-case**, but it avoids exposing syntax objects explicitly, instead relying on just patterns, templates (similar to `#'`-quoted forms), and pattern variables. A **syntax-rules** expression evaluates to a function that takes a syntax object as an argument:

```
> ((syntax-rules ()
     [(_ num) (+ num 1)])
   #'(list 42))
#<syntax:6:0 (+ 42 1)>
```

The ***define-syntax-rule*** form combines implementing a syntax-transforming function in terms of **syntax-rules**, and binding that function as a named macro. Section 2.3.2 has some examples of using **define-syntax-rule**.

In Racket, a macro is defined by creating a *transformer binding* to a syntax-transforming function. For example, we can use ***define-syntax*** to bind `m42` as a macro whose uses expand to syntax for `(list 42)`:

```
> (define-syntax m42 (λ (stx) #'(list 42)))
> (m42)
'(42)
```

Binding `m42` as a transformer means that the name can be referenced within code aimed at run time, but also that the definition's value (e.g., as given by the λ expression above) is already available at macro-expansion time.

Racket's module system is compatible with macros in such a way that it is possible to **provide** macros to export them from a module, or to **require** macros to import them from a module. It is also possible to **provide** and **require** values ***for-syntax***, which makes them available for use at compile time, within macro implementations. For example, we can implement a macro `mπ` that expands to a literal number approximating π:[13]

[13]Similarly to **define**, there is shorthand syntax for binding transformer *functions* with **define-syntax**; i.e., we can write `(define-syntax (mπ stx) ....)` instead of writing out the λ expression explicitly.

```
> (require (for-syntax (only-in racket/math pi)))
> (define-syntax (mπ stx) (datum->syntax stx pi))
> (mπ)
3.141592653589793
```

One of the more Racket-specific macro-system features used in this dissertation is *sub-form expansion*, which allows a macro to expand a sub-form appearing in its input, and then inspect and modify the result of that expansion. A sub-form may be expanded with the ***local-expand*** [Flatt et al., 2012] function, which is quite versatile in terms of what it is able to expand, and to what extent. Section 2.3.4 shows its use to expand essentially an entire module body, but it can also be used to expand a single expression. For example, we can write a macro `swap-branches` that expands its input up to any **`if`** forms, and then swaps the "then" and "else" branches of an **`if`** form appearing immediately in the expansion result. In this example, we assume that an expression such as `(and 1 2)` expands roughly as `(if 1 2 #f)`:

```
> (define-syntax (swap-branches stx)
    (with-syntax ([(_ sub) stx])
      (define x (local-expand #'sub 'expression (list #'if)))
      (syntax-case x (if)
        [(if c thn els) #'(if c els thn)])))
> (swap-branches (and 1 2))
#f
> (swap-branches (and #f 2))
2
```

Domain uses Macros can help not only language engineers, but also "end programmers" in taking charge of their destiny, and moulding their languages on demand as target platforms evolve and domain selections change. Macros have many potential applications in a niche platform product line, such as: extending languages with domain-specific features; implementing other, higher level, more specialized compile-time mechanisms (e.g., conditional compilation or mapped types); or allowing the programmer to code compile-time computations for better run-time efficiency. If side effects during macro expansion are acceptable, one might even generate auxiliary resource files required for implementing an abstraction.

There are cases where it is desirable for a programming language to have features too specific for being built into a general-purpose language. For example, the Qt framework (as of version 5) relies on custom C++ extensions for its own events mechanism and run-time introspection support, and implements them in terms of C++ preprocessor macros and an external *`moc`* (Meta-Object Compiler) code analyzer and generator; if C++ had a more powerful macro language, such external code generation might be unnecessary.

A sufficiently powerful macro-enabled language can also serve as a basis for implementing other, less general compile-time mechanisms. As an example, we might want Magnolisp to have language for conditional compilation in the style of the C preprocessor's **`#if`** .... **`#elif`** .... **`#else`** .... **`#endif`** construct. To achieve that, we might simply use Racket as the conditional expression

language, and define a *static-cond* construct as a macro:[14]

```
(define-syntax (static-cond stx)
  (syntax-case stx (else)
    [(_)
     ; default suitable for definition (but not expression) contexts
     #'(begin)]
    [(_ [else thn ...])
     ; explicit default
     #'(begin thn ...)]
    [(_ [c thn ...] . more)
     ; evaluate condition c at compile time
     (if (syntax-local-eval #'c)
         #'(begin thn ...)
         #'(static-cond . more))]))
```

Armed with `static-cond` in our language, we could then check a compile-time configuration (with its variables imported **`for-syntax`** to make them available at compile time) to pick different run-time code depending on the target platform. For example:

```
(require (for-syntax "config.rkt"))

(define (init-any-ui w)
  (static-cond
   [(or on-bb10 on-harmattan on-sailfish)
    (init-qt-ui w)] ; use Qt for targets that have it
   [on-console
    (init-ncurses-ui w)] ; use ncurses for console builds
   [else
    w])) ; headless build (no UI)
```

Programmers often have their applications do more work at run time than is fundamentally required, due to language limitations in what compile-time computations can conveniently (or at all) be expressed. A generic compiler extension mechanism (supporting programmable computation at compile time) such as Racket macros may allow for more performant application designs, which is particularly beneficial for resource-constrained targets.

Again, the programmer may choose what kind of macro-defined language (if any) to use in support of more "static" designs. Examples of potentially useful language include: `static-cond`; Magnolia-style static component language; C++ style **`constexpr`** definitions, for constant expressions to evaluate at compile time; or domain-specific sub-languages whose expressions *might* evaluate partially, which may also lead to more accurate static analysis [Herman and Meunier, 2004].

Self-extension A language with macros is self-extensible, and I consider this to be an important (if not essential) characteristic in the context of niche platform PLE. Some of the benefits include:

[14] In Racket, a "line comment" begins with a semicolon, and continues until the end of the line.

- Self-extensibility allows delegating language engineering work from the production tools developer to the domain programmer.
- Voelter and Visser [2011] point out that it is good to allow language to be defined incrementally and iteratively, with user validation after each iteration, and delegating language engineering work to the user (the domain programmer) encourages such iterative development.
- A self-extension facility enables localized syntactic abstraction. Racket macros' templates can even close over bindings in a local context, which for example makes it possible for a macro use to implicitly operate on certain local variables.

1.5 Product-Line Development Kits

One of my principles for assembling a cross-niche-platform "development kit" for a product line is to combine adaptable, target-platform-agnostic tooling with platform-vendor-provided tooling. This combination is appropriate as there may not *be* any third-party-provided, platform-specific tools in a niche setting. The principle calls for flexibility from the target-agnostic tool infrastructure, so that it can be made to interface with the appropriate vendor tools.

A comprehensive set of tooling includes things such as: a (deployable) programming language, assisted editing and code navigation support, simulated and on-target testing and debugging support, etc. Mainstream software developers in particular have come to expect such features, creating pressure for platform vendors to provide them. For each programmable platform we might minimally expect to be provided with at least one programming language, an application programming interface (API), and on-device deployment support. For the rest of a comprehensive product-line development kit, any common tooling would ideally be self-sufficient; in the case of this dissertation, that tooling revolves around a mouldable programming language and its language environment.

It would be burdensome for a small independent software vendor to create both a language (possibly in multiple different flavors) and its development environment from scratch. One way to avoid that is to use a language workbench such as Spoofax [Kats and Visser, 2010], JetBrains MPS[15], or Xtext[16] in defining the language, as the same definitions then work as a basis for IDE support.

Charles et al. [2009] suggest that language-specification-derived support may not be flexible enough for IDE experiences that fully reflect distinctive language characteristics; Magnolia takes a slightly lower level approach, and instead bases its Eclipse IDE integration on the *IMP* framework [Charles et al., 2009], which provides building blocks for standard services such as syntax highlighting, hover help, and outline view [Bagge, 2013].

Magnolisp adopts yet another approach in that it arranges for direct reuse of another language's tooling, through language embedding into Racket. This means that it may not be necessary to implement any supporting tools for the language at all, if one is willing to live without a language-optimized

[15] `https://www.jetbrains.com/mps/`
[16] `http://www.xtext.org/`

development experience. One advantage of the embedding approach is the availability of editing services that—unlike those produced by established language workbenches—can cope with self-extensible languages; a caveat is that it is harder to implement satisfactory refactorings for such languages, if their results should be resugared [Pombrio and Krishnamurthi, 2015] for readability. Further benefits of language embedding into Racket are discussed in section 2.5.

Existing Racket code editing facilities tend to work fine for Magnolisp. For instance, racket-mode[17] for the Emacs text editor supports Magnolisp code navigation, auto-completion, and documentation lookup, without any Magnolisp-specific configuration, while the DrRacket IDE [Findler et al., 2002] is able to visualize Magnolisp binding structure:

```
5 (typedef Int #:: (foreign))
6
7 (declare (add1 x)
8  #:: (foreign [type (-> Int Int)]))
```

Racket's Scribble [Flatt et al., 2009] language and compiler can be used for typesetting and rendering documentation for Magnolisp APIs. Racket's `raco make` tool can be used as a build manager for Magnolisp bytecode, to speed up further compilation to C++; this should be particularly beneficial for relatively stable libraries that are common dependencies.

DrRacket's macro debugger can be used to debug Magnolisp macros, since the macro system is Racket's. Magnolisp supports API mocking and execution "simulation" within the Racket VM, and thus some testing and debugging of code is possible without going via C++; this can also be done in a Magnolisp REPL (i.e., an interactive console), as launchable via the `racket` executable. For testing cross-compiled code on the target, it is necessary to rely on vendor tools, which often include support for target platform emulation on the workstation, or on-target debugging through a target device connection. Axiom-based testing can also be seen as a form of debugging support as it can help pinpoint errors for fixing, without running an application as such, but rather unit testing its code via a test harness to find operations to blame for breaking asserted invariants.

1.6 Target Languages, APIs, and Systems

In this section I briefly discuss some recent and current niche smartphone operating systems and cross-platform development technologies.

When considering available smartphone platform developer offerings, one may notice that large vendors tend to favor their own proprietary frameworks (e.g., Cocoa Touch and .NET Compact Framework), and even maintain their own languages (like Swift or C#) and associated tools. This may be because large vendors can afford to fund the development of such technology, and because platform-specific technologies help achieve a degree of vendor lock-in. Niche platforms seem to be more commonly built in terms of languages and libraries that competing vendors are also using (C++ and Qt, for example). Thus, even if niche platform tools do not receive as much investment or atten-

[17] `https://github.com/greghendershott/racket-mode` (2016)

tion as their mainstream counterparts, there may be more native code reuse opportunities across them.

1.6.1 Niche Smartphone Operating Systems

I define *smartphone* as a mobile phone that allows for third-party development and installation of applications that run natively, i.e., in the same execution environment as the platform's built-in applications and services generally do. In this section I discuss some smartphone operating systems from the point of view of a native application developer; section 4.2 adds to the discussion by comparing the permission-based security models of a number of smartphone systems.

Overall, most (if not all) smartphones today support C and C++ for development, perhaps in part because many games developers insist on them. C++ support is sometimes incomplete, however; the full standard library may not be available, for example, or the run-time library may lack support for some language features. OpenGL ES is also almost ubiquitous, and likewise widely used in games and other graphics-intensive applications. Other near ubiquitous libraries include SQLite (for databases) and zlib (for data compression). A smartphone SDK commonly includes a version of GCC or Clang (or sometimes both) as a cross-compiler, possibly with some customizations (e.g., for binary compatibility). It is particularly common for an SDK to include an Eclipse or Qt Creator based IDE, or a plugin for the Microsoft Visual Studio IDE.

BlackBerry 10 (BB10) is a smartphone OS based on the QNX real-time, microkernel-based OS. While QNX, like Linux, is POSIX compliant, there nonetheless are differences that may hamper portability. As of version 10.0.9, the BB10 native SDK includes: the C and C++ standard libraries, OpenGL ES, EGL, SQLite, cURL, and Qt, in addition to platform-specific APIs; QCC as a GCC-based compiler for targeting a QNX-based system; QNX Momentics as a standalone Eclipse-based IDE; and a plugin for the Microsoft Visual Studio IDE.

Tizen[18] (as of version 2.4) is an operating system with a Linux kernel, and RPM as its package manager; it has shipped on smartphones and other consumer devices. The Tizen SDK includes an Eclipse-based IDE, and both GCC and LLVM based toolchains for compilation. Tizen's selection of cross-platform libraries includes: the C and C++ standard libraries, OpenGL ES, OpenMP, SQLite, libxml, and zlib.

Sailfish OS[19] (as of version 2.0) is a mobile OS based on Mer, a mobile-optimized "core Linux" distribution using RPM for package management; Qt is supported as an API, and the SDK uses Qt Creator as its IDE, and VirtualBox for Sailfish emulation on a workstation.

MeeGo Harmattan (or just *Harmattan*) is a smartphone OS derived from the Maemo operating system, which in turn is based on Debian, and thus uses dpkg as its package manager. Unlike earlier Maemo releases, Harmattan has a Qt-based UI; in addition to Qt, the platform includes the standard C and C++ libraries, OpenGL ES, GLib, GStreamer, and PulseAudio. A Qt-based SDK

[18] `https://www.tizen.org/`

[19] `https://sailfishos.org/`

was the primary application developer offering, including Qt Creator as the IDE. There was also a separate Harmattan Platform SDK for (direct) access to underlying platform functionality; it did not include an IDE.

Symbian OS is written in C++ from the kernel up, and—like the QNX-based BB10—it is also a rare example of a smartphone OS with a microkernel architecture [Sales, 2005]. During Symbian's history the SDK offerings varied, but most recently there were both Eclipse and Qt Creator centric SDKs, and API support included the C and C++ standard libraries, OpenGL ES, OpenVG, EGL, Boost, SQLite, and Qt.

Symbian is a peculiar smartphone OS in that it was widely used by consumers but disliked by developers, perhaps due to being so unlike other systems. Kantee and Vuolteenaho [2006] found that—compared to UNIX and Windows—Symbian is a "completely different kind of system," and that assumptions which hold in other environments may no longer be valid. Symbian originates from a time of modest hardware specifications and C++ before its standardization. Consequently, it goes to great lengths to be resource frugal, mostly at the developers' expense, and its programming idioms can seem weird; the weirdnesses of the "Symbian C++" dialect have been summarized by Pagonis [2007]. I give examples of Symbian idioms in briefly describing a Symbian-specialized programming language in section 6.9.3.

1.6.2 Cross-Platform Application Frameworks

Niche platforms may be more likely to ship with more widely used application frameworks than their mainstream competition, but this reuse benefit is offset by there being more third-party cross-platform abstraction layer offerings for mainstream ecosystems (e.g., React Native[20] 0.20 only supports Android and iOS targets).

While such abstraction layers open up significant opportunities for code reuse across platforms, they also have their own drawbacks:

- They tend to be costly to maintain (depending on their implementation strategy), and may drop support for a platform with little notice, or even get discontinued altogether; this risk should naturally be weighed against the risk of the underlying platforms themselves getting discontinued, or having their native frameworks replaced by something else.
- A framework that attempts to abstract over various platform aspects may easily fall far behind an evolving platform in what portion of the underlying functionality it manages to cover.
- A platform-abstraction layer may be large, with its use resulting in runtime overhead that harms the user experience (e.g., by noticeably increasing application launch times).
- A framework may have a prohibitively large footprint for bundling with every small application, and in that case it perhaps cannot be used unless the target platform has it built-in, or has a facility for shared installation (e.g., a package manager).

[20] `https://facebook.github.io/react-native/`

- A target platform vendor might even impose contractual obligations prohibiting the use of certain kinds of third-party frameworks in their sales channels.
- A cross-platform framework's UI widgets might not look and feel quite the same as the underlying platform's (which may not always be considered a drawback).

One may try to judge a framework's potential longevity based on its implementation strategy. Cross-platform frameworks that build on widely available APIs and limit exposure to proprietary APIs should be less costly to maintain, and thus more likely to survive for longer periods of time. Perhaps the most prominent example of an API that has become nearly ubiquitous in modern PCs and smartphones is OpenGL (or OpenGL ES) for graphics rendering. Qt 5.0's core GUI module (i.e., "Qt GUI") requires OpenGL, for example, while the popular Unity[21] game engine uses OpenGL for some of its target platforms[22].

Web technologies are another proven substrate for cross-platform application implementation. One strategy for exploiting them is to embed a web view for rendering UIs (using any available API for such embedding), and to expose a selection of platform services to any JavaScript, HTML, and CSS code that runs within the view. *Apache Cordova*[23] (formerly *PhoneGap*) is a widely used application framework that employs this strategy, and over time it has supported an unusually large variety of niche platforms such as bada, BB10, Firefox OS, Tizen, Ubuntu Touch, and webOS. This support attests to the portability of this approach, but a large part of the value of Cordova is in its ecosystem of "plugins" that provide access to platform functionality.

A flexible cross-niche-platform product-line strategy is not to rely on the availability of cross-platform frameworks or rich supporting ecosystems, but to interface or incorporate such technology where that is advantageous. In general, one can probably get the most out of a platform with the least overhead and deployment difficulty by using platform-native APIs directly, but that may not always be the most cost-effective option. Fortunately, in some cases, one can get both native and cross-platform at once, as there are niche platforms that adopt a cross-platform framework as their native application framework.

The long-lived *Qt*[24] application framework in particular has been well represented on niche platforms. BB10, Harmattan, Sailfish, Sharp Zaurus, and Ubuntu Touch, for example, are (or were) natively based on Qt, while later versions of S60 also included and made some use of Qt in their UI and applications. Qt 5.5 is also installable on Android, iOS, and Windows Phone, among others, even though none of those platforms have Qt APIs built-in. Qt's wide cross-platform support—which includes major desktop OSes—makes it sort of a "meta" platform that has a larger developer community than many of its host systems.

As of version 5.5, Qt is C++ based, and exceptional in being comprehensive enough to host entire applications; depending on an application's functionality, it may not require direct access to any underlying operating system

[21] `http://unity3d.com/` (2016)

[22] Unity 5.3.3's supported smartphone targets include Android, iOS, Windows Phone, and Tizen. BB10 was supported by Unity versions 4.2–5.1.4.

[23] `http://cordova.apache.org/` (2016)

[24] `http://www.qt.io/`

APIs. The resulting portability of Qt-based software is valuable, also enabling Qt-based software targeted for smartphones to be at least partially tested on workstations.

Idiomatic Qt C++ code is object-oriented and stylistically similar to Java. Exceptions and RTTI (run-time type information) are not used, and there is no extensive use of C++ templates [Qt Wiki, 2016].

Qt extends C++ through the use of the `moc` tool, adding support for object introspection, among other things. The Qt object model is reliant on `moc`-generated code, and this introduces some complications, particularly to code organization and build management. Firstly, in a mixed Qt/non-Qt codebase, one will probably want to invoke `moc` on Qt APIs only, requiring some bookkeeping about which files to `moc`. Secondly, one typically ends up with a larger number of header files than normally in C++, in arranging for `moc` to be invoked also on *private* Qt APIs. Thirdly, to avoid errors in `moc`cing one must be mindful of any interactions with the C++ preprocessor, particularly with codebases that have optional, compile-time configurable features (`moc`'s support for preprocessing varies depending on the Qt version).

A transcompiled programming language targeting C++ can avoid any problems related to invoking `moc` by generating code that already includes the definitions that `moc` would generate. I briefly describe such a Qt-specialized language in section 6.9.2.

1.6.3 C++ as a Niche Platform Lingua Franca

C++ plays a special role as our primary transcompilation target language, with Magnolia, Magnolisp, and Erda all having a C++ back end. For a language like Magnolia, supporting parameterized programming [Goguen, 1984], C++ is an attractive target due to its parameterizability: as the C++ template mechanism enables compile-time parameterization of code, it becomes possible to do component composition *after* source-to-target language translation. I discuss that further in section 6.4.1, which concerns interfacing with foreign languages from a component-system-equipped language.

In this dissertation, a specific attraction of C++ is its wide reach over niche smartphone platforms. While C is still quite entrenched as a language of choice in embedded software development [Voelter et al., 2015], C++ presently strikes a good balance between smartphone platform toolchain support and its compatibility with APIs, since both C and C++ APIs are directly accessible from C++. Some smartphone platform APIs (e.g., bada and Symbian) are also natively C++. A C++ code asset that builds on top of a popular API such as the C++ Standard Library or Qt should be usable across a number of potential target platforms, which is valuable from a domain-engineering point of view.

Targeting the common subset of C and C++ would give even better platform coverage; the Lua 5.0 VM, for example, is highly portable due to being implemented in that subset [Ierusalimschy et al., 2005]. Transcompilation permits us to generate idiomatic C++, however, without costing us the possibility to add support for additional target languages (e.g., C or Objective-C) as necessary for toolchain or API compatibility.

By adopting a transcompilation-based portability strategy, any *C++ back end* may liberally exploit the richness of its target language for readability and type safety, for example, as I imply in section 6.9.1 by discussing a more

C++-oriented variant of Magnolisp. As I further suggest in sections 6.9.2 and 6.9.3 by considering additional variants of that language, one could even support targeting of different idiomatic *styles* of C++. Such support might be beneficial for portability or API use convenience, as C++ is a large "multi-paradigm" language arguably lacking a de-facto coding standard. Each C++ sub-community (bada, Boost, Qt, Symbian, etc.) uses its own language subset with its own stylistic conventions.

The kind of C++ that one is expected to use depends not only on conventions, but also the language version, and any vendor-specific deviations from the specification. If a platform's toolchain lacks support for a C++ feature, then it can of course not be used when targeting that platform, with transcompilation or otherwise. For example, C++ RTTI did not become available on Symbian OS before version 9, and no version of bada supports C++ exceptions. Exceptions, multiple inheritance, and templates are among the features commonly missing from C++ compilers for embedded systems [White, 2011].

Non-standard language extensions may also be a source of incompatibilities, and both Clang and GCC, for instance, have a number of them for C++. Such extensions may also originate from platform vendors; for example, Apple's OS X v10.6 Xcode developer tools (which can also build for iOS) introduced a form of closures called "Blocks" [Garst, 2009], allowing them to be used in C, C++, and Objective-C.

1.7 A Niche Platform Software Production Strategy

> "Now the general who wins a battle makes many calculations in his temple ere the battle is fought."
>
> Sun Tzu, as translated by Lionel Giles

This dissertation's high-level strategy for constructing and running a niche-platform-compatible software product line is characterized by these key design choices:

- **Core asset management:** Manage software components, target-specific build information, and program configurations in an open-world and tool-assisted manner.
- **Production tooling:** Have tooling complement and integrate with a programming language family sharing a common style and development environment. Accommodate target-platform-specific tools, but avoid depending on them.
- **Program composition:** Have at least one of the language family members include a component system supporting parameterizable components, and separate abstract "concepts" that may specify both syntax and semantics (as signatures and algebraic laws) to constrain composition. Have the language possess a static reasoning and translation friendly algorithmic language, to support configuration-fact inference and cross-platform targeting. Potentially have this platform-agnostic composition language include syntax to support any coding conventions for dealing with cross-cutting concerns (such as error and event handling).

- **Language engineering:** Include a language-family-wide mechanism for language extension, to serve as common ground for implementing and evolving the family, and specializing languages for different purposes (e.g., domain-conceptual expression or tools integration). Have extensions be modular, thus also turning them into manageable and reusable assets. Ideally support language self-extension, and allow extensions to be composable and scoped, to support safer pervasive use of syntactic abstraction and piecemeal co-evolution of language and code. Build languages as towers on top of a small core, for maintainability, but do allow inherently different languages to have different cores.
- **Programming:** Program business logic portably, against concepts as platform abstractions. Implement components natively in terms of target frameworks and libraries, in languages that specialize in interfacing with target APIs directly.
- **Deployment:** Source-to-source compile programs into target-native languages for further building and deployment with platform-specific tools. Compile to human-readable source code to facilitate debugging.

1.8 Outline

I now present an overview of the remaining chapters of this dissertation. For the chapters that are not solely authored by me, I will indicate what part of the contribution is mine:

- **Chapter 2** describes a technique for infrastructure reuse for a compiler front end. The technique concerns exploiting Racket's language definition machinery to enable separate source-to-source compilation of the implemented languages, with various potential infrastructure reuse benefits for the source language. The chapter is a full-length version of a published paper co-authored by Matthew Flatt, whose technical expertise was essential in discovering the technique, and particularly its use of "submodules," which were new at the time. The rough idea for the paper was mine. I implemented the Magnolisp programming language, in part for purposes of validating the presented technique.
- **Chapter 3** describes a scheme for implementing program transformation infrastructure with compiler "middle ends" and intermediate program representations in mind. More specifically, the scheme is for macro-based, language-integrated abstract syntax tree API generation, with some unconventional support for defining abstractions over related abstract syntax. The chapter is an unpublished paper co-authored by my co-supervisor Anya Helene Bagge. I am the main author, and the idea was mine. I created the Illusyn program transformation library, which includes an abstract syntax API generator implemented according to the described scheme.
- **Chapter 4** describes an application of language technology to the domain of permission-based security models. It explains how one might keep track of applications' permission requirements in a cross-platform setting, by exploiting a programming language that is both equipped with a component system, and designed to allow accurate static reasoning.

The chapter is a published paper that I co-authored with Anya Helene Bagge and my supervisor Magne Haveraaen. I am the main author, and the original idea was mine. Designing the syntax and semantics of permission declarations for the Magnolia programming language was a joint effort. Permission declaration and inference support was implemented for Magnolia by Anya. I implemented the original version of the Anyxporter example application, while Anya adapted it for Magnolia.

- **Chapter 5** discusses another application of language technology. It presents a convention for failure management that is suitable for a cross-platform setting, as it can be realized in highly portable code. The chapter also presents language support for the convention, featuring semi-automated reporting, handling, and propagation of errors, as well as a facility for adapting to other conventions. The chapter is a longer and differently-themed version of a soon-to-be published paper that I co-authored with Magne Haveraaen. I am the main author, and worked out many of the semantic details of the language constructs, building on earlier work by the group on syntax and semantics of "alerts" [Bagge et al., 2006]. The original idea for how to represent errors was Magne's, as was the formulation of the relationship to the underlying theory of guarded algebras [Haveraaen and Wagner, 2000]. I created the Erda family of programming languages, which include built-in error-handling support, as described.
- **Chapter 6** sketches a design for using the prior chapters' technologies and language features as part of a larger niche-platform-friendly product-line setup for developing software product families. Unlike the solutions of the earlier chapters, the overall solution sketched here has not been validated through implementation.
- **Chapter 7** provides additional context for this dissertation by discussing some of the benefits, drawbacks, and uncertainties that I perceive in my software production tooling strategy. The chapter also discusses related work with respect to that strategy.
- **Chapter 8** is the conclusion.

CHAPTER 2

Adopting a Macro System

If we want our programming language to be translatable into multiple vendor-supported languages, we can make the translation work easier by avoiding sophisticated features that lack counterparts in the target languages. We might instead have sophisticated language facilities for syntactic abstraction, which can then be used to translate abstract code into some fairly primitive language that is easy to translate further.

Macros are a classic language facility for enabling programmers to define custom syntactic abstractions. For pervasive abstraction in large-scale software, it is perhaps best to support programmable, module-system-aware hygienic macros, for purposes of expressiveness, safer composition, and more modular (and reusable) organization. Macro systems with all of those characteristics can be complex, and our primitive, transcompiled language may not be the best basis for macro programming.

In this chapter we describe one way to reuse an existing language's macro system for a language of our own, by embedding our language within that existing host language.

By designing our language to incorporate an existing macro system and macro language and its libraries, we gain extensibility for our source language, with the extensions expressible in a language that is free of any target language constraints. Furthermore, we will be able to benefit from existing languages' ecosystems and tools both at the source-language side (e.g., for documenting, editing, and extending) and at the target-language side (e.g., for building, deploying, and testing).

I presented an early version of this paper at IFL 2014 in Boston [Hasu and Flatt, 2014], and a condensed version at ELS 2016 in Kraków [Hasu and Flatt, 2016].

Source-to-Source Compilation via Submodules

Tero Hasu
BLDL and
University of Bergen

Matthew Flatt
PLT and
University of Utah

Abstract

Racket's macro system enables language extension and definition primarily for programs that are run on the Racket virtual machine, but macro facilities are also useful for implementing languages and compilers that target different platforms. Even when the core of a new language differs significantly from Racket's core, macros offer a maintainable approach to implementing a larger language by desugaring into the core. Users of the language gain the benefits of Racket's programming environment, its build management, and even its macro support (if macros are exposed to programmers of the new language), while Racket's syntax objects and submodules provide convenient mechanisms for recording and extracting program information for use by an external compiler. We illustrate this technique with Magnolisp, a programming language that runs within Racket for testing purposes, but that compiles to C++ (with no dependency on Racket) for deployment.

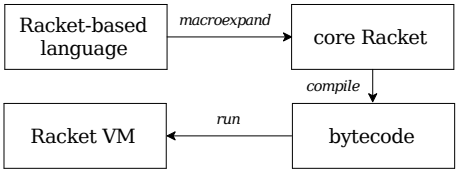
2.1 Introduction

A *macro expander* supports the extension of a programming language by translating extensions into a predefined core language. A *source-to-source compiler* (or *transcompiler* for short) is similar, in that it takes source code in one language and produces source code for another language. Since both macro expansion and source-to-source compilation entail translation between languages, and since individual translation steps can often be conveniently specified as macro transformations, a macro-enabled language can provide a convenient platform for implementing a transcompiler.

Racket's macro system, in particular, not only supports language extension—where the existing base language is enriched with new syntactic forms—but also language *definition*—where a completely new language is implemented though macros while hiding or adapting the syntactic forms of the base language. The Racket macro system is thus suitable for implementing a language with a different or constrained execution model relative to the core Racket language.

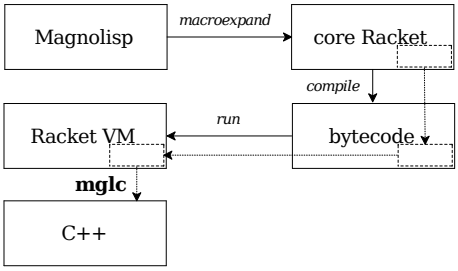
Magnolisp is a Racket-based language that targets embedded devices. Relative to Racket, Magnolisp is constrained in ways that make it more suitable for platforms with limited memory and processors. For deployment, the Magnolisp compiler transcompiles a core language to C++. For development, since cross-compilation and testing on embedded devices can be particularly time consuming (compilation times generally pale in comparison to the time used to transfer, install, and launch a program), Magnolisp programs also run directly on the Racket virtual machine (VM) using libraries that simulate the target environment.

Racket-based languages normally target only the Racket VM, where macros expand to a core Racket language, core Racket is compiled into bytecode form, and then the bytecode form is run:



To instead transcompile a Racket-based language, Magnolisp could access the representation of a program after it has been macro-expanded to its core (via the **`read`** and **`expand`** functions). Fully expanding the program, however, would produce Racket's core language, instead of Magnolisp's core language. External expansion would also miss out on some strengths of the Racket environment, including automatic management of build dependencies.

Magnolisp demonstrates an alternative approach that takes full advantage of Racket mechanisms to assemble a "transcompile time" view of the program. The macros that implement Magnolisp arrange for a representation of the core program to be preserved in the Racket bytecode form of modules. That representation can be extracted as input to the `mglc` compiler to C++:



In this picture, the smaller boxes correspond to a core-form reconstruction that is only run in transcompile-time mode (as depicted by the longer arrow of the "run" step). The boxes are implemented as submodules [Flatt, 2013], and the core form is extracted by running the submodules instead of the main program modules.

By compiling a source program to one that constructs an AST for use by another compiler layer, our approach for Magnolisp in Racket is similar to lightweight modular staging in Scala [Rompf and Odersky, 2010] or strategies that exploit type classes in Haskell [Chakravarty et al., 2011]. Magnolisp demonstrates how macros can achieve the same effect, but with the advantages

of macros and submodules over type-directed overloading: more flexibility in defining the language syntax, support for static checking that is more precisely tailored to the language, and direct support for managing different instantiations of a program (i.e., direct evaluation versus transcompilation).

2.2 Magnolisp

Magnolisp[1] is statically typed, and all data types and function invocations are resolvable to specific implementations at compile time. Static typing for Magnolisp programs facilitates compilation to efficient C++, as the static types can be mapped directly to their C++ counterparts. To reduce syntactic clutter from annotations and to help retain untyped Racket's "look and feel," Magnolisp supports type inference à la Hindley-Milner.

Magnolisp's surface syntax is similar to Racket's syntax for common constructs, but it also has language-specific constructs, including ones that do not directly map into Racket core language (e.g., **`if-cxx`** for conditional transcompilation). Magnolisp uses Racket's module system for managing bindings, both for run-time functions and for macros. An exported C++ interface is defined separately through ***`export`*** annotations on function definitions; only **`export`**ed functions are declared in the generated C++ header file.

A Magnolisp module starts with `#lang magnolisp`. The module's top-level can ***`define`*** functions, types, and so on. A function marked as ***`foreign`*** is assumed to be implemented in C++; it may also have a Racket implementation, given as the body expression, to allow it to be run in the Racket VM. Types defined in C++ are also **`foreign`**, and ***`typedef`*** can be used to give the corresponding Magnolisp declarations. The ***`type`*** annotation is used to specify types for functions and variables, and type expressions can refer to **`typedef`**-bound type names. The ***`#::`*** keyword is used to specify a set of annotations for a definition. A shorthand macro supports the declaration of multiple ***`primitives`*** in terms of **`foreign`**, `#::`, and so on.

In the following example, `get-last-known-location` is a Magnolisp function of type `(-> Loc)`, i.e., a function that returns a value of type `Loc`. The `(rkt.get-last-known-location)` expression in the function body could be a call to a Racket function from module `"positioning.rkt"` to simulate position-information retrieval:

```
#lang magnolisp
(require (prefix-in rkt. "positioning.rkt"))

(typedef Loc #:: (foreign))

(define (get-last-known-location)
  #:: (foreign [type (-> Loc)])
  (rkt.get-last-known-location))
```

No C++ code is generated for the above definitions, as they are both declared as **`foreign`**. For an example with a C++ translation, consider this code, which uses Magnolisp's Racket-style **`let`** and **`if`** expressions:

[1] Documentation: `https://bldl.ii.uib.no/software/pltnp/magnolisp.html`

```
#lang magnolisp
(require magnolisp/std/list)

; element primitives (defined in C++)
(typedef Int #:: ([foreign int]))
(define (add x y)
  #:: (foreign [type (-> Int Int Int)]))

; sum of first two list elements (or fewer for shorter lists)
(define (sum-2 lst) #:: (export)
  (if (empty? lst)
      0
      (let ([t (tail lst)])
        (if (empty? t)
            (head lst)
            (add (head lst) (head t))))))
```

The transcompiler-generated C++ implementation for the `sum-2` function is the following (apart from minor reformatting):

```
MGL_API_FUNC int sum_2( List<int> const& lst ) {
  List<int> t;
  return is_empty(lst) ?
      0 :
      ((t = tail(lst)),
       (is_empty(t) ? head(lst) :
         add(head(lst), head(t))));
}
```

We could exploit macros in defining functions like `sum-2`. We do so with the following left-folding `mfold` macro, which performs macro-expansion-time unrolling of recursion. This time the **`let`** and **`if`** forms appear within a code-generation template:

```
; uses op to combine at most n first elements of lst
; (result is init for an empty list)
(define-syntax (mfold stx)
  (syntax-case stx ()
    [(_ op init lst n)
     (let ([i (syntax-e #'n)])
       (if (= i 0)
           #'init
           #`(let ([tmp lst])
              (if (empty? tmp)
                  init
                  (op (head tmp)
                      (mfold op init (tail tmp)
                             #,(sub1 i))))))))]))

(define (sum-2/2 lst) #:: (export)
  (mfold add 0 lst 2))
```

whose `sum-2/2` function gets translated by `mglc` as

```
MGL_API_FUNC int sum_2_2( List<int> const& lst ) {
  List<int> tmp;
  return is_empty(lst) ?
      0 :
      add(head(lst),
          ((tmp = tail(lst)),
           (is_empty(tmp) ? 0 : add(head(tmp), 0))));
}
```

Figure 2.1 shows an overview of the Magnolisp architecture, including both the `magnolisp`-defined front end and the `mglc`-driven middle and back ends. Figure 2.2 illustrates the forms of data that flow through the compilation pipeline. Transcompilation triggers running of `"a.rkt"` module's transcompile-time code, through `magnolisp-s2s` submodule's instantiation by invoking ***`dynamic-require`*** to fetch values for certain variables (e.g., `def-lst`); the values describe the code of `"a.rkt"`, and are already in the compiler's internal data format. Any referenced dependencies of `"a.rkt"` (e.g., `"num-types.rkt"`, as indicated by `int`'s binding information) are processed in the same manner, and the relevant definitions are incorporated into the compilation result (i.e., `"a.cpp"` and `"a.hpp"`).

The middle and back ends are accessed either via the *`mglc`* command-line tool or via the underlying API as a Racket module. In either case, the expected input is a set of modules for transcompilation into C++. The compiler loads any transcompile-time code in the modules and their dependencies. Any module with a `magnolisp-s2s` submodule is assumed to be Magnolisp, but other Racket-based languages may also be used for macro programming or simulation. The Magnolisp compiler effectively ignores any code that is not run-time code in a Magnolisp module.

The program transformations performed by the compiler are generally expressed with term-rewriting strategies. These strategies are implemented by a custom combinator library[2] that is inspired by Stratego [Bravenboer et al., 2008]. Syntax trees that are prepared for the transcompilation phase instantiate data types that support the primitive strategy combinators of the combinator library.

The compiler middle end implements whole-program optimization (by dropping unused definitions), type inference, and some simplifications (e.g., removal of condition checks where the condition is constant). The back end implements translation from Magnolisp core to C++ syntax (involving, e.g., lambda lifting), copy propagation, C++-compatible identifier renaming, splitting of code into sections (e.g.: public declarations, private declarations, and private implementations), and pretty printing.

2.3 Hosting a Transcompiled Language in Racket

Magnolisp is an example of a general strategy for building a transcompiled language within Racket. In this section, we describe some details of that process for an arbitrary transcompiled language *L*. Where the distinction matters,

[2] `https://bldl.ii.uib.no/software/pltnp/illusyn.html`

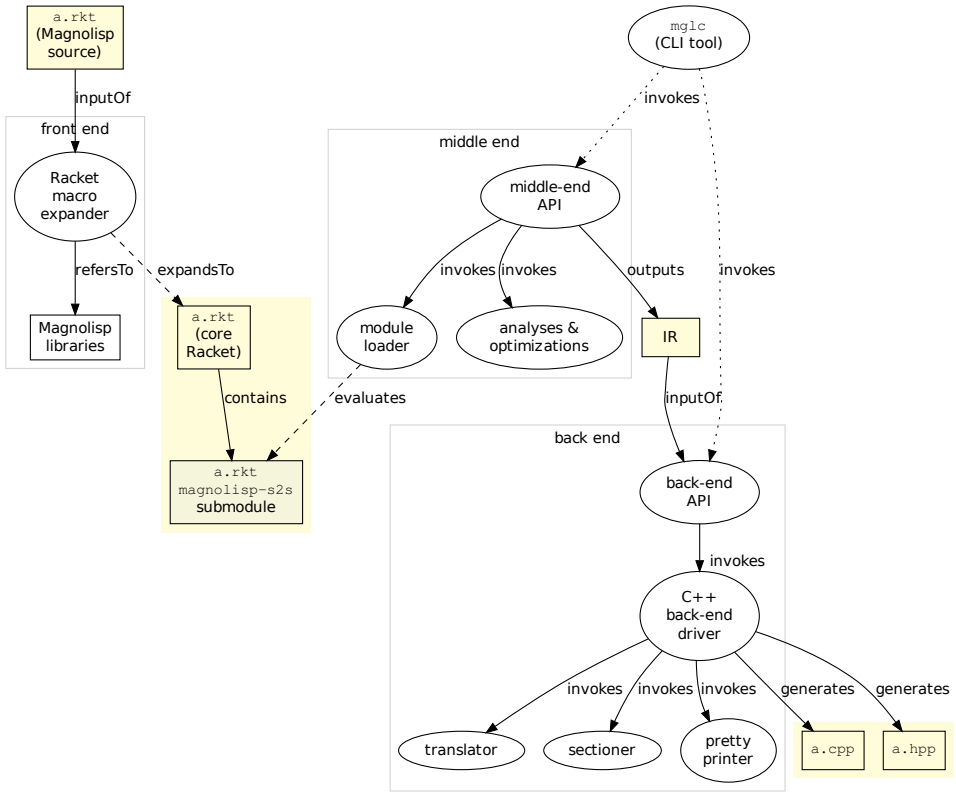


Figure 2.1: The overall architecture of the Magnolisp implementation, showing some of the components involved in compiling a Magnolisp source file `"a.rkt"` into a C++ implementation file `"a.cpp"` and a C++ header file `"a.hpp"`. The dotted arrows indicate that the use of the `mglc` command-line tool is optional; the middle and back end APIs may also be invoked by other programs. The dashed "evaluates" arrow indicates a conditional connection between the left and right hand sides of the diagram; the `magnolisp-s2s` submodule is *only* loaded when transcompiling. The "expandsTo" connection is likewise conditional, as `"a.rkt"` may have been compiled to bytecode ahead of time, in which case the module is already available in a macro-expanded form; otherwise it is compiled on demand by Racket.

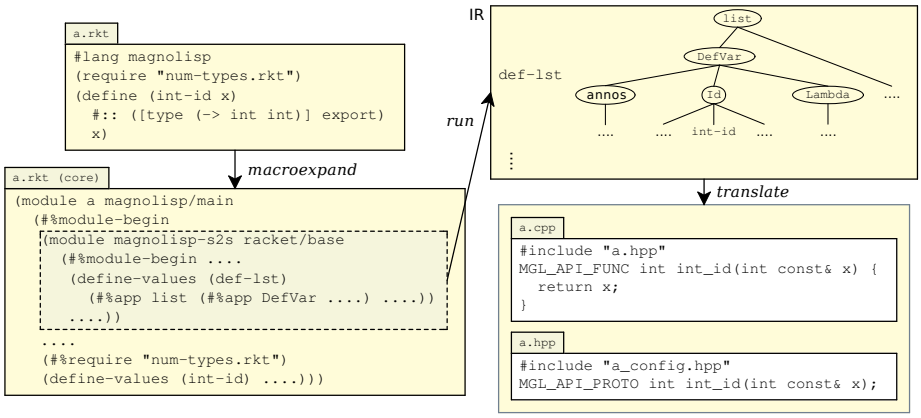


Figure 2.2: Subset of Figure 2.1 showing file content: a Magnolisp module passing through the compilation pipeline.

we use L_R to denote a language that is intended to also run in the Racket VM (possibly with mock implementations of some primitives), and L_C to denote a language that only runs through compilation into a different language.

Building a language in Racket means defining a module or set of modules to implement the language. The language's modules define and export macros to compile the language's syntactic forms to core forms. In our strategy, furthermore, the expansion of the language's syntactic forms produces nested submodules to separate code than can be run directly in the Racket VM from information that is used to continue compilation to a different target.

2.3.1 Modules and `#lang`

All Racket code resides within some *module*, and each module starts with a declaration of its *language*. A module's language declaration has the form `#lang` *L* as the first line of the module. The remainder of the module can access only the syntactic forms and other bindings made available by the language *L*.

A language is itself implemented as a module.[3] A language module is connected to the name *L*—so that it will be used by `#lang` *L*—by putting the module in a particular place in the filesystem or by appropriately registering the module's parent directory. In general, a language's module provides a *reader* that gets complete control over the module's text after the `#lang` line. A reader produces a *syntax object*, which is a kind of S-expression (that combines lists, symbols, etc.) that is enriched with source locations and other lexical context. We restrict our attention here to using the default reader, which parses module content directly as S-expressions, adding source locations and an initially empty lexical context.

[3]Some language must be predefined, of course. For practical purposes, assume that the `racket` module is predefined.

For example, to start the implementation of *L* such that it uses the default reader, we might create a `"main.rkt"` module in an `"L"` directory, and add a `reader` submodule that points back to `L/main` as implementing the rest of *L*:

```
#lang racket
(module reader syntax/module-reader L/main)
```

The S-expression produced by a language's reader serves as input to the macro-expansion phase. A language's module provides syntactic forms and other bindings for use in the expansion phase by exporting macros and variables. A language *L* can re-export all of the bindings of some other language, in which case *L* acts as an extension of that language, or it can export an arbitrarily restrictive set of bindings. For example, if `"main.rkt"` re-exports all of `racket`, then `#lang L` is just the same as `#lang racket`:

```
#lang racket
(module reader syntax/module-reader L/main)
(provide (all-from-out racket))
```

A language must at least export a macro named *`#%module-begin`*, because that form implicitly wraps the body of a module. Most languages simply use **`#%module-begin`** from `racket`, which treats the module body as a sequence of ***`require`*** importing forms, ***`provide`*** exporting forms, definitions, expressions, and nested submodules, where a macro use in the module body can expand to any of the expected forms. A language might restrict the body of modules by either providing an alternative `#%module-begin` or by withholding other forms. A language might also provide a `#%module-begin` that explicitly expands all forms within the module body, and then applies constraints or collects information in terms of the core forms of the language.

As an example, the following `"main.rkt"` re-exports all of `racket` except **`require`** (and the related core language name **`#%require`**), which means that modules in the language *L* cannot import other modules. It also supplies an alternate `#%module-begin` macro to pre-process the module body in some way:

```
#lang racket
(module reader syntax/module-reader L/main)
(provide
  (except-out (all-from-out racket)
              require #%require #%module-begin)
  (rename-out [L-module-begin #%module-begin]))
(define-syntax L-module-begin ....)
```

For transcompilation, the `#%module-begin` macro plays a key role in our strategy. A Racket language *L* that is intended for transcompilation is defined as follows:

- The language's module exports bindings that define the surface syntax of the language. The provided bindings expand only to transcompiler-supported run-time forms. We describe this step further in section 2.3.2
- Where applicable, macros record additional metadata that is required for transcompilation. We describe this step further in section 2.3.3

- The `#%module-begin` macro fully expands all the macros in the module body, so that the rest of the transcompiler pipeline need not implement macro expansion. We describe this step further in section 2.3.4
- After full macro expansion, `#%module-begin` adds externally loadable information about the expanded module into the module. We describe this step further in section 2.3.5
- Any run-time support for running programs is provided alongside the macros that define the syntax of the language. We describe this step further in section 2.3.6

The export bindings of L may include variables, and the presence of transcompilation introduces some nuances into their meaning. When the meaning of a variable in L is defined in L, we say that it is a *non-primitive*. When its meaning is defined in the execution language, we say that it is a *primitive*. When the meaning of its appearances is defined by a compiler (or a macro) of L, we say that it is a *built-in*. As different execution targets may have different compilers, a built-in for one target may be a primitive for another.

2.3.2 Defining Surface Syntax

A module that implements the surface syntax of a language L exports a binding for each predefined entity of L, whether that entity is a built-in variable, a core-language construct, or a derived form. When the core language is a subset of Racket, derived forms obviously should expand to the subset. Where the core of L is a superset of Racket, additional constructs need an encoding in terms of Racket's core forms where the encoding is recognizable after expansion; possible encoding strategies include:

- **E1**. Use a variable binding to identify a core-language form, and use it in an application position to allow other forms to appear within the application form. Subexpressions within the form can be delayed with suitable **`lambda`** wrappers, if necessary.
- **E2**. Attach information to a syntax object through its *syntax property* table (as found in every syntax object); macros that manipulate syntax objects must then propagate properties correctly.
- **E3**. Store information about a form in a compile-time table that is external to the module's syntax objects.
- **E4**. Use Racket core forms that are not in L (not under their original meaning), or combinations of forms involving such forms.

A caveat for **E2** and **E3** is that syntax properties and compile-time tables are transient, generally becoming unavailable after a module is fully expanded; any information to be preserved must be reflected as generated code in the module's expansion, as discussed in section 2.3.5. Another caveat of such "out-of-band" storage is that identifiers in the stored data must not be moved out of band too early; a binding form must be expanded before its references are moved so that each identifier properly refers to its binding.

In the case of L_R, the result of a macro-expansion should be compatible with both the transcompiler and the Racket evaluator. The necessary duality can be achieved if the surface syntax defining macros can adhere to these constraints: (**C1**) exclude Racket core form uses that are not supported by the

compiler; (**C2**) add any compilation hints to Racket core forms in a way that does not affect evaluation (e.g., as custom syntax properties); and (**C3**) encode any compilation-specific syntax in terms of core forms that appear only in places where they do not affect Racket execution semantics.

Where constraints **C1**–**C3** cannot be satisfied, a fallback is to have `#%module-begin` rewrite either the run-time code, transcompile-time code, or both, to make the program conform to expected core language. Note that rewriting may still be constrained by the presence of binding forms.[4]

For cases where a language's forms do not map neatly to Racket binding constructs, Racket's macro API supports explicit *definition context*s [Flatt et al., 2012], which enable the implementation of custom binding forms that cooperate with macro expansion.

For an example of foreign core form encoding strategy **E1**, consider an L_C with a `parallel` construct that evaluates two forms in parallel. This construct might be defined simply as a "dummy" constant, recognized by the transcompiler as a specific built-in by its identifier, translating any appearances of `(parallel e1 e2)` "function applications" appropriately:

```
(define parallel #f)
```

Alternatively, as an example of strategy **E2**, L_C's `(parallel e1 e2)` form might simply expand to `(list e1 e2)`, but with a `'parallel` syntax property on the `list` call to indicate that the argument expressions are intended to run in parallel:

```
(define-syntax (parallel stx)
  (syntax-case stx ()
    [(parallel e1 e2)
     (syntax-property #'(list e1 e2) 'parallel #t)]))
```

For L_R, `parallel` might instead be implemented as a simple pattern-based macro that wraps the two expressions in `lambda` and passes them to a `call-in-parallel` run-time function, again in accordance to strategy **E1**. The `call-in-parallel` variable could then be treated as a built-in by the transcompiler and implemented as a primitive for running in the Racket VM:

```
(define-syntax-rule (parallel e1 e2)
  (call-in-parallel (lambda () e1) (lambda () e2)))
```

For an example of adhering to constraint **C3**, we give a simplified definition of Magnolisp's `typedef` form. A declared type `t` is bound as a variable to allow Racket to resolve type references; these bindings also exist for evaluation as Racket, but they are never referenced at run time. The `#%magnolisp` built-in is used to encode the meaning of the variable, but as it has no useful definition in Racket, evaluation of any expressions involving it is prevented. The `CORE` macro is a convenience for wrapping `(#%magnolisp ....)` expressions in an

[4] The principal constraint on encoding a language's form is that a binding form in L should be encoded as a binding form in Racket, because bindings are significant to the process of hygienic macro expansion. Operations on a fully expanded module's syntax objects, furthermore, can reflect the accumulated binding information, so that a transcompiler may possibly avoid having to implement its own management of bindings.

(**if** #f .... #f) form to "short-circuit" the overall expression and make it obvious to the Racket bytecode optimizer that the enclosed expression is never evaluated. The ***annotate*** form is a macro that stores the annotations a **...**, which might, for example, include t's C++ name.

```
(define #%magnolisp #f)

(define-syntax-rule (CORE kind arg ...)
  (if #f (#%magnolisp kind arg ...) #f))

(define-syntax-rule (typedef t #:: (a ...))
  (define t
    (annotate (a ...) (CORE 'foreign-type))))
```

Using a macro system for syntax definition offers several advantages compared to parsing in a more traditional way:[5]

- When custom syntactic forms can be defined as macros, parsing is almost "for free." At the same time, the ability to customize a language's reader makes it possible for surface syntax to be different from Lisp's parenthesized, prefix notation.
- Macros and the macro API provide a convenient implementation for desugaring and other rewriting-based program transformations. Such transformations can be written in a modular and composable way.
- For making *L* itself macro-extensible, the implementation of *L* can simply expose a selection of relevant Racket constructs—directly or through macro adapters—to enable the inclusion of compile-time code within *L* modules.

2.3.3 Storing Metadata

A language implementation may involve *metadata* that describes a syntax object, but is not itself a core syntactic construct in the language. Such data may encode information (e.g., optimization hints) that is meaningful to a compiler or other kinds of external tools. Metadata might be collected automatically by the language infrastructure (e.g., source locations in Racket), it might be inferred by macros at expansion time, or it might be specified as explicit *annotations* in source code (e.g., the **export** annotation of Magnolisp functions, or the **weak** modifier of variables in the Vala language).

There is no major difference between encoding foreign syntax in terms of Racket core language or encoding metadata; strategies **E1**–**E4** apply for both. The main way in which metadata differs is that it does not tend to appear (or at least not remain) as a node of its own in a syntax tree. Any annotations in *L* do have surface syntax, and thus appear explicitly in source code, but such code cannot in general be directly analyzed, as unexpanded *L* code cannot be parsed. A more workable strategy is to have *L*'s syntactic forms store any necessary metadata during macro expansion.

[5] A macro's process of validating and destructuring its input syntax can also be regarded as parsing, even though the input is syntax objects rather than raw program text or token streams [Culpepper, 2012].

For metadata, storage in syntax properties is a typical choice. Typed Racket, for example, stores its type annotations in a custom `'type-annotation` syntax property [Tobin-Hochstadt et al., 2011].

Compile-time tables are another likely option for metadata storage. For storing data for a named definition, one might use an *identifier table*, which is a dictionary data structure where each entry is keyed by an identifier. An *identifier*, in turn, is a syntax object for a symbol. Such a table is suitable for both local and top-level bindings, because the syntax object's lexical context can distinguish different bindings that have the same symbolic name.

Recording metadata in compile-time state has the specific advantage of the data getting collated already during macro expansion which enables lookups across macro invocation sites, without any separate program analysis phase. One could, for example, keep track of variables annotated as `#:mut`able, perhaps to enforce legality of assignments already at macro-expansion time, or to declare immutable variables as **`const`** in C++ output:[6]

```
(define-for-syntax mutables (make-free-id-table))

(define-syntax (my-define stx)
  (syntax-case stx ()
    [(_ x v)
     #'(define x v)]
    [(_ #:mut x v)
     (free-id-table-set! mutables #'x #t)
     #'(define x v)]))
```

It is also possible to encode annotations in the syntax tree proper, which has the advantage of fully subjecting annotations to macro expansion. Magnolisp adopts this approach for its **`annotate`** form for annotation recording, which translates to a special `'annotate`-property-flagged **`let-values`** form to contain annotations. Each contained annotation expression `a` (e.g., `[type ....]`) has its Racket evaluation prevented by encoding it as a Magnolisp **`CORE`** form. Magnolisp's **`type`** and **`annotate`** forms are defined roughly as follows:

```
; type annotation with type expression t
(define-syntax-rule (type t)
  (CORE 'anno 'type t))

; annotates the enclosed expression e
(define-syntax (annotate stx)
  (syntax-case stx ()
    [(_ (a ...) e)
     (syntax-property
      (syntax/loc stx ; retain stx's source location
        (let-values ([() (begin a (values))] ...)
          e))
      'annotate #t)]))
```

The **`annotate`**-generated **`let-values`** forms introduce no bindings, and their right-hand-side expressions yield no values; only the expressions themselves

[6] The ***`define-for-syntax`*** form is like **`define`**, but for macro-expansion time.

matter. Where the annotated expression `e` is an initializer expression, the Magnolisp compiler decides which of the annotations to actually associate with the initialized variable.

2.3.4 Expanding Macros

One benefit of reusing the Racket macro system with *L* is to avoid having to implement an *L*-specific macro system. When the Racket macro expander takes care of macro expansion, the remaining transcompilation pipeline only needs to understand *L*'s core syntax (and any related metadata). Racket includes two features that make it possible to expand all the macros in a module body, and afterwards process the resulting syntax, all within the language.

The first of these features is the `#%module-begin` macro, which can transform the entire body of a module. The second is the **`local-expand`** [Flatt et al., 2012] function, which may be used to fully expand all the `#%module-begin` sub-forms. Using the two features together is demonstrated by the following macro skeleton, which might be exported as the `#%module-begin` of a language:

```
(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(module-begin form ...)
     (let ([ast (local-expand
                 #'(#%module-begin form ...)
                 'module-begin null)])
       (do-some-processing-of ast))]))
```

The **`local-expand`** operation also supports *partial* sub-form expansion, as it takes a "stop list" of identifiers that prevent descending into sub-expressions with a listed name. At first glance, one might imagine exploiting this feature to allow foreign core syntax to appear in a syntax tree, and simply prevent Racket from proceeding into such forms. The main problem with this strategy is that foreign binding forms would not be accounted for in Racket's binding resolution. That problem is compounded if foreign syntactic forms can include Racket syntax sub-forms; the sub-forms need to be expanded along with enclosing binding forms. Indeed, to prevent these problems, **`local-expand`** (in most cases) automatically extends a stop list to include additional Racket core forms if it includes anything, so that partial expansion is constrained to the consistent case that stays outside of binding forms.

2.3.5 Exporting Information to External Tools

After the `#%module-begin` macro has fully expanded the content of a module, it can gather information about the expanded content to make it available for transcompilation. The gathered information can be turned into an expression that reconstructs the information, and that expression can be added to the overall module body that is produced by `#%module-begin`.

The expression to reconstruct the information should *not* be added to the module as a run-time expression, because extracting the information for transcompilation would then require running the program (in the Racket VM).

Instead, the information is better added as compile-time code. The compile-time code is then available from the module while compiling other *L* modules, which might require extra compile-time information about a module that is imported into another *L* module. More generally, the information can be extracted by running only the compile-time portions of the module, instead of running the module normally.

As a further generalization of the compile-time versus run-time split, the information can be placed into a separate *submodule* within the module. A submodule can have a dynamic extent (i.e., run time) that is unrelated to the dynamic extent of its enclosing module, and its bytecode may even be loaded separately from that of the enclosing module. As long as a compile-time connection is acceptable, a submodule can include syntax-quoted data that refers to bindings in the enclosing module, so that information can be easily correlated with bindings that are exported from the module.

For example, suppose that *L* implements definitions by producing a normal Racket definition for running within the Racket virtual machine, but it also needs a syntax-quoted version of the expanded definition to compile to a different target. The ***module+*** form can be used to incrementally build up a `to-compile` submodule that houses definitions of the syntax-quoted expressions:

```
(define-syntax (L-define stx)
  (syntax-case stx ()
   [(L-define id rhs)
    (with-syntax ([rhs2 (local-expand #'rhs
                         'expression null)])
     #'(begin
         (define id rhs2)
         (begin-for-syntax
           (module+ to-compile
             (define id #'rhs2)))))]))
```

Wrapping the `to-compile` submodule with ***begin-for-syntax*** makes it reside at compilation time relative to the enclosing module, which means that loading the submodule will not run the enclosing module. Within `to-compile`, the expanded right-hand side is quoted as syntax using `#'`.

Syntax-quoted code is often a good choice of representation for code to be compiled again to a different target language, because lexical-binding information is preserved in a syntax quote. Source locations are also preserved, so that a compiler can report errors or warnings in terms of a form's original location (`mglc` fetches original source text based on location).

Another natural representation choice is to use any custom intermediate representation (IR) of the compiler. Magnolisp, for example, processes Racket syntax trees already during macro expansion, turning them into its IR format, which also incorporates metadata. The IR uses Racket **`struct`** instances to represent AST nodes, while still retaining some of the original Racket syntax objects as metadata, for purposes of transcompile-time reporting of semantic errors. Magnolisp programs are parsed at least twice, first from text to Racket syntax objects by the reader, and then from syntax objects to the IR by **`#%module-begin`**; additionally, any macros effectively parse syntax objects to syntax objects. As parsing is completed already in **`#%module-begin`**, any

Magnolisp syntax errors are discovered even when just evaluating programs as Racket.

The `#%module-begin` macro of `magnolisp` exports the IR via a submodule named `magnolisp-s2s`. The submodule contains an expression that reconstructs the IR, albeit in a somewhat lossy way, excluding details that are irrelevant for compilation. The IR is accompanied by a table of identifier binding information indexed by module-locally unique symbols, which the transcompiler uses for cross-module resolution of top-level bindings, to reconstruct the identifier binding relationships that would have been preserved by Racket if exported as syntax-quoted code. As `magnolisp-s2s` submodules do not refer to the bindings of the enclosing module, they are loadable independently from it.

2.3.6 Run-Time Support

The modules that implement a Racket language can also define run-time support for executing programs. For L, such support may be required for the compilation target environment; for L_R, any support would also be required for the Racket VM. Run-time support for L is required when L exports bindings to run-time variables, or when the macro expansion of L can produce code referring to run-time variables (even if such a variable's run-time existence is very limited, as it is for `#%magnolisp`).

Every run-time variable requires a run-time binding, so that Racket can resolve references to them. When binding built-ins and primitives of L_C, any initial value expression can be given, as the expressions are not evaluated. A literal constant expression is a suitable initializer for built-ins of L_R, which are initialized for Racket VM execution, but generally never referenced.

Each non-primitive is—by definition—implemented in L, with a single definition applicable for all targets. Strictly speaking, though, any non-primitive that is *exported* by a Racket module L *cannot* itself be implemented in L, but must use a smaller language; the Racket module system does not allow cyclic dependencies.

Defining a primitive of L involves specifying a translation for appearances of the variable into any target language. For a Racket VM target, the variable's value must specify its meaning. For other targets, it may be most convenient to specify the target language mapping in L, assuming that L includes specific language for that purpose. As the mappings are only needed during transcompilation, any metadata specifying them might be placed into a module that is only loaded on demand by the compiler.

The `magnolisp` language, for example, binds three run-time variables, all of which are built-ins. Of these, `#%magnolisp` is only used for its binding, and only during macro expansion. The compiler knows that conditional expressions must always be of type `Bool`, and that `Void` is *the* unit type of the language; this knowledge is useful during type checking and optimization. References to the Magnolisp built-ins may appear in code generated by `magnolisp`'s macros, and hence they must already be bound for the language implementation. Their metadata (specifying C++ translations) is not required by the macros, however, which makes it possible to `declare` that information separately, using Magnolisp's own syntax for storing metadata for an existing binding:

```
#lang magnolisp/base
(require "core.rkt" "surface.rkt")
(declare #:type Bool #:: ([foreign bool]))
(declare #:type Void #:: ([foreign void]))
```

2.4 Evaluation

Our Racket-hosted transcompilation approach is generic—in theory capable of accommodating a large class of languages. In practice, we imagine that it is most useful for hosting newly developed languages (such as Magnolisp), where design choices can achieve a high degree of reuse of the Racket infrastructure. In particular, Racket's support for creating new, extensible languages could be a substantial motivation to follow our approach. Racket hosting is particularly appropriate for an evolving language, since macros facilitate quick experimentation with language features.

Another potential use of our strategy is to add transcompilation support for an existing Racket-based language. We have done so for Erda[7], creating *Erda$_{C++}$* as its C++-translatable variant. Erda has Racket-like syntax, but its evaluation differs significantly from both Racket and Magnolisp. Erda$_{C++}$ programs nonetheless compile to C++ using an unmodified Magnolisp compiler.

Erda$_{C++}$ illustrates that Magnolisp is not only a language, but also infrastructure for making Racket-based languages translatable into C++. A *Magnolisp-based language* must be transformable into Magnolisp's core language, which is more limited than that of Racket (lacking first-class functions, escaping closures, etc.), but the language can have its own runtime libraries (whose names must be `magnolisp-s2s`-communicated to `mglc`). The Racket API of Magnolisp includes a ***make-module-begin*** function that makes it convenient for other languages to implement `mglc`-compatible `#%module-begin` macros—ones that communicate all the expected information.

A potential drawback of transcompilation is the disconnect between the original, unexpanded code and its corresponding transcompiler-generated source code. This disconnect can make debugging difficult when things go wrong. The problem is made worse by macros, and it can be particularly pressing when the output is hard for humans to read. Since Racket's macro expansion preserves source locations, however, a transcompiler could at least emit the original locations via **`#line`** directives (as in C++) or source maps (as supported by some JavaScript environments).

2.4.1 Language Design Constraints

In our experience, two design constraints make Racket reuse especially effective: the hosted language's name resolution should be compatible with Racket's, and the hosted language's syntax should use S-expressions.

Overloading as a language feature, for instance, appears a bad fit for Racket's name resolution. Instead of overloading, names in Racket programs are typically prefixed with a datatype name, as in **`string-length`** and **`vector-length`**. Constructs for renaming at module boundaries, such as ***`prefix-in`*** and ***`prefix-out`***, help implement and manage name-prefixing conventions.

[7] `https://bldl.ii.uib.no/software/pltnp/erda.html`

An S-expression syntax is not strictly necessary, but Racket's macro programming APIs work especially well with its default parsing machinery. The language implementor can then essentially use concrete syntax in patterns and templates for matching and generating code. This machinery is comparable to concrete-syntax support in program transformation toolkits such as Rascal [Klint et al., 2009] and Spoofax [Kats and Visser, 2010]. Still, other kinds of concrete syntaxes can be adopted for Racket languages, with or without support for expressing macro patterns in terms of concrete syntax, as demonstrated by implementations of Honu [Rafkind and Flatt, 2012] and Python [Ramos and Leitão, 2014].

2.4.2 Example Use Case: A Static Component System

Macro-enabled extensibility combined with a constrained core (as in Magnolisp) provides an opportunity to explore the limits of macro-based expressiveness. We sketch a use case for capable macros in a constrained context: a component system in Magnolisp.

When organizing a collection of software building blocks, it can be useful to have a mechanism for "wiring up" and parameterizing building blocks to form larger wholes (e.g., individual software products of a product line). Racket already includes a component system of *units* [Culpepper et al., 2005; Owens and Flatt, 2006], which are first-class, dynamically composed components.

The macro-expansion of Racket's unit construct uses features of Racket's core that are not included in Magnolisp. However, at compile time Magnolisp has access to all of Racket, and hence enough power to implement a purely static component system. As a proof of concept, we provide an implementation of a rudimentary component system in figure 2.3.

Existing solutions suggest that it should also be possible to implement a more capable static component system in terms of Racket macros. Chez Scheme's **`module`**s support static, external linking, and have been shown to cater for a variety of use cases [Waddell and Dybvig, 1999]. Racket's built-in "packages" system resembles the Chez design, and is implemented in terms of macros, relying on features such as sub-form expansion, definition contexts, and compile-time binding. As packages are implemented statically, they require little from the run-time language.

2.5 Motivation for Racket-Hosted Transcompilation

Hosting a language via macros offers the potential for extensibility in the hosted language. This means leveraging the host language both to provide a language extension mechanism and a language for programming any extensions. While a basic language extension mechanism (such as the C preprocessor or a traditional Lisp macro system) may be implementable with reasonable effort, safer and more expressive mechanisms require substantial effort to implement from scratch. Furthermore, supporting programmable (rather than merely substitution based) language extensions calls for a compile-time language evaluator, which may not be readily available for a transcompiled language.

Hosting in Racket offers safety and composability of language extensions through lexical scope and phase separation respecting macro expansion.

```
#lang magnolisp
(define-syntax-rule (define<> x f e)
  (define-syntax f (cons #'x #'e)))

(define-syntax (use stx)
  (syntax-case stx (with as)
    [(_ f with new-x as fx)
     (let ([v (syntax-local-value #'f)])
       (with-syntax ([old-x (car v)] [e (cdr v)])
         #'(define fx
             (let-syntax ([old-x
               (make-rename-transformer #'new-x)])
               e))))]))

(primitives [#:type int] [#:type long]
  [#:function (->long x) :: (-> int long)])

(define<> T id
  (annotate ([type (-> T T)])
    (lambda (x) x)))

; int int_id(int const& x) { return x; }
(use id with int as int-id)
; long long_id(long const& x) { return x; }
(use id with long as long-id)

; long run(int const& x)
;   { return long_id(to_long(int_id(x))); }
(define (run x) #:: (export)
  (long-id (->long (int-id x))))
```

Figure 2.3: A primitive "component" system for Magnolisp. The macro `define<>` declares a named "expression template" `f`, and the macro `use` specializes such templates for a specific parameter `x`. Use of the two macros is demonstrated by a C++-inspired function template `id` with a type parameter `T`, also showing how macros might compensate for lack of parametric polymorphism. Corresponding `mglc`-generated C++ code is given in comments.

Racket macros' hygiene and referential transparency help protect the programmer from inadvertent "capturing" of identifiers, making it more likely that constructs defined modularly (or even independently) compose successfully. *Phase separation* [Flatt, 2002] means that compile time and run time have distinct bindings and state. The separation in the time dimension is particularly crucial for a transcompiler, as it must be possible to parse code without executing it. The separation of bindings, in turn, helps achieve language separation, in that one can have Racket bindings in scope for compile-time code, and hosted-language bindings for run-time code.

Racket's handling of modules can also be leveraged to support modules in the hosted language, with Racket's `raco make` tool for rebuilding bytecode then automatically serving as a build tool for multi-module programs in the language. The main constraint is that Racket does not allow cycles among module dependencies.

Particularly for new languages it can be beneficial to reuse existing language infrastructure. With a Racket embedding one is in the position to reuse Racket infrastructure on the front-end side, and the target language's infrastructure (typically libraries) on the back-end side. Reusable front-end side language tools might include IDEs [Findler et al., 2002], documentation tools [Flatt et al., 2009], macro debuggers [Culpepper and Felleisen, 2007], etc. Although some tools might not be fully usable with programs that cannot be executed as Racket, the run vs. compile time phase separation means that a tool whose functionality does not entail running a program should function fully.

Racket's language extension and definition machinery may be useful not only for users, but also for language implementors. Its macros have actually become a compiler front end API that is sufficient for implementing many general-purpose abstraction mechanisms in a way that is indistinguishable from built-in features [Culpepper and Felleisen, 2006]. In particular, a basic "sugary" construct is convenient to implement as a macro, as both surface syntax and semantics can be specified in one place.

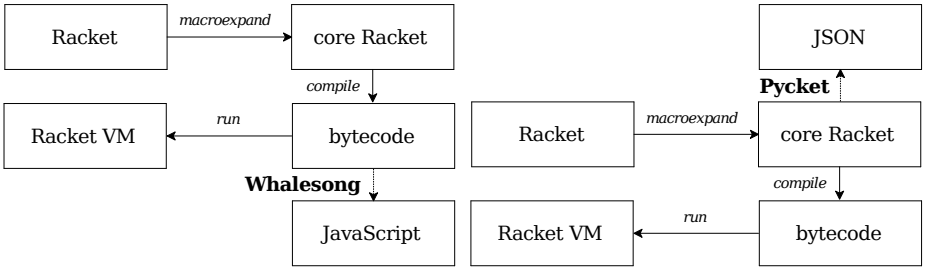
2.6 Related Work

Many language implementations run on Lisp dialects and also target other environments. Some languages, such as Linj [2013] or Clojure plus ClojureScript [2016], primarily provide a Lisp-like language in the target environment. Other languages, such as *STELLA* [Chalupsky and MacGregor, 1999] and Parenscript [2016], primarily match the target environment's semantics but enable execution in a Lisp as well. Magnolisp is closer to the latter group, in that it primarily targets the target environment's semantics.

Most other languages previously implemented on Racket have been meant for execution only on the Racket VM, but a notable exception is Dracula [Eastlund, 2012], which compiles macro-expanded programs to ACL2. Its (so far largely undocumented) compilation strategy is to expand syntactic forms to a subset of Racket's core forms, and to specially recognize applications of certain functions (such as `make-generic`) for compilation to ACL2. The part of a Dracula program that runs in Racket is expanded normally, while the part to be translated to ACL2 is recorded in a submodule through a combination

of structures and syntax objects, where binding information in syntax objects helps guide the translation.

Whalesong [Yoo and Krishnamurthi, 2013] and Pycket [Bauman et al., 2015] are both implementations *of* Racket targeting foreign language environments. Their approaches to acquiring fully macro-expanded Racket core language differ from ours. Whalesong compiles to JavaScript via Racket bytecode, which is optimized for efficient execution (e.g., through inlining), but does not retain all of the original (core) syntax; thus, it is not the most semantics-rich starting point for translation into foreign languages. The Pycket compiler instead performs external expansion to get core Racket; it **`read`**s, **`expand`**s, and JSON-serializes Racket syntax, in order to pass it over to the RPython meta-tracing framework:



Ziggurat [Fisher and Shivers, 2008]—also built on Racket (then PLT Scheme)—is a meta-language system for implementing extensible languages. Its approach allows both for self-extension and transcompilation of languages, with different tradeoffs compared to ours. Ziggurat features hygienic macros that are Scheme-like, but have access to static semantics, as defined for a language through other provided mechanisms; Racket lacks specific support for interleaving macro expansion with custom analysis. Ziggurat's macros may be locally scoped, but not organized into separately loadable modules; Racket allows for both. There is basic safety of macro composition with respect to Ziggurat's own name resolution, but composability of custom static semantics depends on their implementation. Ziggurat includes constructs for defining new syntax object types, while our approach requires encoding "tricks."

Lightweight Modular Staging (LMS) [Rompf and Odersky, 2010] is similar to our technique in goals and overall strategy, but leveraging Scala's type system and overload resolution instead of a macro system. With LMS, a programmer writes expressions that resemble Scala expressions, but the type expectations of surrounding code cause the expressions to be interpreted as AST constructions instead of expressions to evaluate. The constructed ASTs can then be compiled to C++, CUDA, JavaScript, other foreign targets, or to Scala after optimization. AST constructions with LMS benefit from the same type-checking infrastructure as normal expressions, so a language implemented with LMS gains the benefit of static typing in much the same way that a Racket-based language can gain macro extensibility. LMS has been used for languages with application to machine learning [Sujeeth et al., 2011], linear transformations [Ofenbeck et al., 2013], fast linear algebra and other data structure optimizations [Rompf et al., 2012], and more.

The Accelerate framework [Chakravarty et al., 2011; McDonell et al., 2013] is similar to LMS, but in Haskell with type classes and overloading. As with LMS, Accelerate programmers benefit from the use of higher-order features in Haskell to construct a program for a low-level target language with only first-order abstractions.

Copilot [Pike et al., 2013] is also a Haskell-embedded language whose expressions are interpreted as AST constructions. Like Racket, Copilot has a core language, into which programs are transformed prior to execution. The Copilot implementation includes two alternative back ends for generating C source code; there is also an interpreter, which the authors have employed for testing. Copilot's intended domain is the implementation of programs to monitor the behavior of executing systems in order to detect and report anomalies. The monitoring is based on periodic sampling of values from C-language symbols of the monitored, co-linked program. Since such symbols are not available to the interpreter, the language comes built-in with a feature that the programmer may use to specify representative "interpreter values" for any declared external values [Pike et al., 2012]; this is similar to Magnolisp's support for "mocking" of **`foreign`** functions.

The *Terra* programming language [DeVito et al., 2013] takes an approach similar to ours, as it adopts an existing language (Lua) for compile-time manipulation of constructs in the run-time language (Terra). Like Racket, Terra allows compile-time code to refer to run-time names in a way that respects lexical scope. Terra is not designed to support transcompilation, and it compiles to binaries via Terra as a fixed core language. Another difference is Terra's emphasis on supporting code generation at run time, while our emphasis is on separation of compile and run times.

CGen [Selgrad et al., 2014] is a reformulation of C with an S-expression-based syntax, integrated into Common Lisp. An AST for source-to-source compilation is produced by evaluating the CGen core forms; this differs from our approach, where run-time Racket core forms are not evaluated. Common Lisp's **`defmacro`** construct is available to CGen programs for defining language extensions; Racket's lexical-scope-respecting macros compose in a more robust manner. Racket's macro expansion also tracks source locations, which would be a useful feature for a CGen-like tool. CGen uses the Common Lisp package system to implement support for locally and explicitly switching between CGen and Lisp binding contexts, so that ambiguous names are shadowed; Racket does not include a similar facility (approximate implementations thereof should be possible within Racket, however).

SC [Hiraishi et al., 2007] is another reformulation of C with an S-expression-based syntax. It supports language extensions defined by transformation rules written in a separate, Common Lisp based domain-specific language (DSL). The rules treat SC programs as data, and thus SC code is not subject to Lisp macro expansion (as in our solution) or Lisp evaluation (as in CGen). Fully transformed programs (in the base SC-0 language) are compiled to C source code. SC programs themselves have access to a C-preprocessor-style extension mechanism via which there is limited access to Common Lisp macro functionality.

2.7 Conclusion

We have described a generic approach for having Racket host the front end of a source-to-source compiler. The strategy involves a proper embedding of the hosted language into Racket, so that Racket's usual language definition facilities are exploited rather than bypassed. Notably, the macro and module systems are still available and, if exposed to the hosted language, provide a way to implement and manage language extensions within the language. Furthermore, tools such as the DrRacket IDE work with the hosted language, recognize the binding structure of programs written in the language, and can usually trace the origins of macro-transformed code.

Among the various ways to arrange for a source-to-source compiler to gain access to information about a program, our approach is most appropriate when the language's macros target a specific foreign core language and runtime library and when it is useful to avoid "extra-linguistic mechanisms" [Felleisen et al., 2015] by having the language itself communicate its execution requirements to the outside world. Such communications may be prepared as submodules, which can also contain an AST in the appropriate core language and representation, allowing one source language to support multiple different targets. Racket's separate compilation and build management help limit preparation work to modules whose source files or dependencies have changed.

Racket's macro system is expressive enough that the syntax and semantics of a variety of language constructs can be specified in a robust way. Given that typical macros compose safely, and given that hygiene reduces the likelihood of name clashes and allows macros to be defined privately, pervasive use of syntactic abstraction becomes a realistic alternative to manual or tools-assisted writing of repetitive code. Such abstraction can benefit both the codebase implementing a Racket-based language, as well as programs written in a macro-enabled Racket-based language.

Acknowledgements

Carl Eastlund provided information about the implementation of Dracula. Magne Haveraaen, Anya Helene Bagge, and anonymous referees provided useful comments on drafts of this paper. This research has in part been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language.

CHAPTER 3

Abstract Data Representations for Abstract Syntax

In Racket's macro model, S-expressions expand into S-expressions, and ultimately those expressions will be in a core language. In Racket and many other Lisp implementations, core S-expressions are eventually parsed into abstract syntax trees for further processing prior to execution. As explained in chapter 2, we can choose what information to encode into enriched S-expressions, for purposes of source-to-source compilation.

To support generation of human-approachable code in a variety of target languages, we likely want to retain more source-language semantics in our program representation than we would for mere execution; that information needs to be carried through to the individual target language back ends that require it, meaning that we require a wider-than-usual communication path through the compilation pipeline.

In this chapter, we discuss how we might make further use of macros in a compiler implementation, to generate an AST implementation to function as the compiler's intermediate representation. We furthermore describe a way to realize an illusion of an AST representing fewer and simpler syntactic constructs than it actually does, depending on how it is viewed. Our solution is based on AST-node-like interfaces implemented as projections into actual syntax objects. Those underlying data objects are also treated abstractly (as abstract data types), which makes the design especially compatible with interfaces and concepts, as found in Magnolia; this was one of my goals in devising this AST solution.

This chapter gives one answer to the question I presented at OOPSLE 2014 in Antwerp regarding coming up with new ways to declaratively specify abstractions for generated program object models supporting program transformations [Hasu, 2014].

Illusionary Abstract Syntax

Tero Hasu Anya Helene Bagge

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway

Abstract

We present a scheme for declarative abstract syntax implementation such that it: allows for commonality-capturing abstractions beyond common grammar derivations (e.g., common structure or information content); assumes integration with the host language, relying on an expressive macro system instead of an external code generator; and requires little from the generated run-time language, instead shifting responsibility to code-generation time.

Each generated syntax object type exposes an interface, allowing for its data representation to remain hidden if desired. For additional abstraction, one can declare similar abstract interfaces spanning multiple object types or subsets thereof. The distinction between actual and "virtual" objects is further blurred by each interface having an associated "data representation" for purposes of pattern matching.

3.1 Introduction

A compiler translates its source language into its target language through successive program transformation steps, transforming an *intermediate representation* (IR) of some kind. One representation choice is to use an *abstract syntax tree* (AST) to encode source, target, or intermediate language abstract syntax as a tree-like data structure. In an AST implementation, each language construct typically gets a dedicated data type, each following some common data-model-specific conventions.

We have previously [86] speculated that it might be useful to support declarative implementation of abstractions reflecting commonalities other than those implied by the productions of a grammar, e.g., cross-cutting abstract syntax concerns such as nodes containing identifiers, sub-expressions or statement lists. It might furthermore be useful for those abstractions to present the "illusion" of being actual data structures (with a representation). Such abstraction should inherently offer some insulation against incidental changes

in data representation choices, where the abstract syntax itself remains unchanged. It is likewise clear that one can create more general and reusable transformation routines by coding them against abstractions that capture the necessary semantics for a particular kind of transformation, but abstract over related object types where their distinctions are irrelevant.

For example:

- We might have block expressions, if-statements, loops and let-statements all containing lists of statements—and we may only be interested in that list (e.g., for adding/removing parts of it).
- We might have many different kinds of expressions that all have subexpressions that we would like to treat in the same way.
- Some information, such as types, may not be directly part of the AST structure, but we may be able to compute it or look it up, and then present a view of a typed AST.

We present a lightweight AST library to support abstraction of that nature, with simple foundations that: (1) allows the implementation of basic AST abstractions that appear as actual data structures; (2) allows the creation of language abstractions for implementing (or declaring) AST abstractions; and (3) require few language features for describing the run-time behavior of AST implementations. Instead, the scheme requires a capable macro system for implementing language-integrated, domain-specific constructs for specifying the desired data structures and abstractions, and for generating their implementations.

Our design is inspired by the concept of abstract data types: we can generate similar sets of data access operations regardless of whether those operations are for an actual data structure or an abstraction. Furthermore, we can also generate and bind *macros* that create the appearance of abstract data having a concrete representation. Macro uses expand away, however, and any AST access code that remains is not cluttered by the AST implementation details; rather, it just entails abstract values and functions that manipulate them.

We have an implementation of our scheme, incorporated into a small program transformation toolkit that also includes a collection of higher-order functions to assist in programming traversals and transformations over generated ASTs. The toolkit is named Illusyn, and it is implemented as a library in the Racket programming language [70], whose macro system is suitable for performing the necessary code generation within the language.

As an example use case, we have applied Illusyn within the implementation of the research programming language *Magnolisp* [88], using it extensively during analysis and translation into C++ source code. Magnolisp is likewise implemented in Racket, and has depended on macros from the beginning for defining its (extensible) surface syntax. We found it fitting to continue from there by enabling syntactic abstraction of a different nature within the compiler's IR. Magnolisp is also relevant to us in that it demonstrates a way to adopt the Racket macro system for a language that neither is Racket nor requires it for deployment; that is one possibility for acquiring a macro system capable of hosting our AST generation scheme.

3.1.1 Contributions

The main contributions of this paper are:

- a macro-focused scheme for implementing AST APIs, with the benefit of natural host-language embedding, and few requirements on the run-time language, which allows for:
  - declarative implementation of both grammar-derived and cross-cutting non-grammar-derived abstractions for generated abstract syntax trees;
  - unorthodox AST abstractions by adopting views (à la Wadler [189]) for abstract interfaces;
  - view-directed traversals over ASTs;
- Illusyn, a program-transformation library with an AST API generator that implements the above ideas.

3.2 Motivation for Abstraction-Friendly AST APIs

An AST implementation used within a compiler should be able to represent both source and target language programs, and any in-between language. It is common to define an intermediate *core language* with a simpler, somewhat language agnostic syntax, which is something in between "desugared" source language and "ensugared" target language. Programs reduced into core language should have fewer similar-yet-different constructs, requiring fewer cases of alternative processing logic to manipulate; consequently, there should be less repetitive processing code to break when the intermediate representation inevitably changes one day.

Unfortunately, there are limits to core language minimalism. The limitations can be particularly severe in source-to-source compilers; they sometimes require a wider communication path between their front and back ends, so that sufficient semantic information can be carried through the compilation pipeline to allow high-level constructs to be preserved or recreated in the target language. In such cases, the implementor *sometimes* (and only sometimes) needs to treat similar-yet-different constructs differently. This is the primary motivation for our approach: we would like cost-effective implementation of a sufficiently rich AST, with apparent core-language-style simplicity as appropriate.

An abstraction-friendly AST API benefits a compiler engineer in the obvious way of removing the cost of manual implementation of the chosen abstractions (to the extent that the generator supports them). For an example of hand-written code implementing a commonality-capturing abstraction, consider this implementation of a `StatCont` abstraction representing anything containing a sequence of statements:

```
; Statement container is BlockStat, BlockExpr or LetStat
(define (StatCont? ast)
  (or (BlockStat? ast) (BlockExpr? ast) (LetStat? ast)))

; Get the list of statements
(define (StatCont-ss ast)
```

(1)

```
(struct DeclVar (id t))
(struct DefVar (id t v))
```

(2)

```
(struct DeclVar (id t))
(struct DefVar DeclVar (v))
```

(3)

```
(struct DefVar (id t v))
(struct Undefined ())
```

Figure 3.1: Alternative declarations of syntax object types for the same abstract syntax: (1) unrelated types; (2) with subtyping, variable definition being a special case of variable declaration; and (3) with `DeclVar` represented as `DefVar` containing an `Undefined` initializer expression. In each of these definitions, `id` is the name field, `t` is the type field, and `v` is the initial-value-expression field.

```
  (cond
    [(BlockStat? ast) (BlockStat-ss ast)]
    [(BlockExpr? ast) (BlockExpr-ss ast)]
    [(LetStat? ast) (LetStat-ss ast)]
    [else (raise-argument-error
           'StatCont-ss "StatCont?" ast)]))

; Set the list of statements
(define (set-StatCont-ss ast ss)
  (cond
    [(BlockStat? ast) (set-BlockStat-ss ast ss)]
    [(BlockExpr? ast) (set-BlockExpr-ss ast ss)]
    [(LetStat? ast) (set-LetStat-ss ast ss)]
    [else (raise-argument-error
           'set-StatCont-ss "StatCont?" ast)]))
```

The `StatCont?`, `StatCont-ss`, and `set-StatCont-ss` functions together define a small "statement container" interface, abstracting over the differences between language constructs containing statement sequences (e.g., `BlockStat`). The three functions are a predicate and accessors for getting and setting (with functional update) the contained statement sequence `ss`.

Being able to implement such code declaratively should make it less tempting for the engineer to avoid the abstraction adoption cost, and instead resort to writing repetitive, representation-specific program transformations. One might also expect to gain *some* insulation from changes to incidental, implementation-specific program representation choices, such as shown in figure 3.1.

Deciding between concrete object representations can seem arbitrary when no choice is clearly "better" than the other. Equipped with a sufficiently expressive API generator one may be in a position to make the less arbitrary choice of exposing all of the "good" "representations." Such a tool can also make the distinction between object fields and *annotations* (i.e., open-ended

collections of secondary, "non-structural" information in syntax objects) less prominent.

3.3 The Illusyn Library

Illusyn[1] (Illusionary Syntax) is a Racket library for defining AST APIs and implementing them in a mostly declarative manner. As the resulting APIs support term *rewriting strategies* [180], Illusyn can in itself also be regarded as a lightweight program transformation toolkit. In this section we introduce Illusyn, give an overview of its features, and describe the more traditional program transformation features that it has.

Illusyn is founded on the scheme described later in section 3.8—for each construct in the abstract syntax, we provide a node type and functions to access and manipulate the node (analogous to the way one might make one class for each construct in an object-oriented AST implementation). On top of those foundations we have built a number of AST abstraction facilities:

- Specifically, we allow each concrete node type to have any number of abstract interfaces, each of which may abstract over any number of concrete node types (perhaps only concerning parts of the data in them);
- we optionally allow only a subset of a node type's values to support an abstract interface;
- we allow conformance to one abstraction to imply conformance to others;
- we make abstract interfaces resemble those of concrete nodes, supporting pattern matching, (prototype-based) building, and traversal, among other operations;
- we adopt the views mechanism (à la Wadler [189]) for pattern matching on the abstract interfaces, which by design have exactly one "natural" view (that the programmer need not specify).

As data representation for AST nodes, Illusyn uses Racket *structures*, which are instances of *structure types*. Such types are record types with named *fields*, making them a form of algebraic data type (of the "product type" kind). However, Racket's structure types also have some of the characteristics of abstract data types (ADTs), since they are given a public name and some type-specific operations (such as a membership predicate and an accessor function for each field). This benefits us in that we can create the illusion of abstract interfaces appearing as if they were regular Racket structure types; we discuss this further in section 3.4.

A structure type may be defined by the user with the **`struct`** form. Such a definition also binds a type-specific *constructor* with positional arguments corresponding to the order of the declared fields. The **`struct`** form also "co-operates" [73] with Racket's pattern matching construct (i.e., **`match`** [168]), so that structures may be matched by their type name, and their fields by their constructor position. Again, through Racket's extension facilities, we can create the illusion of abstract interfaces being data structures with positional fields; we discuss this further in section 3.5.

[1] Documentation: `https://bldl.ii.uib.no/software/pltnp/illusyn.html`

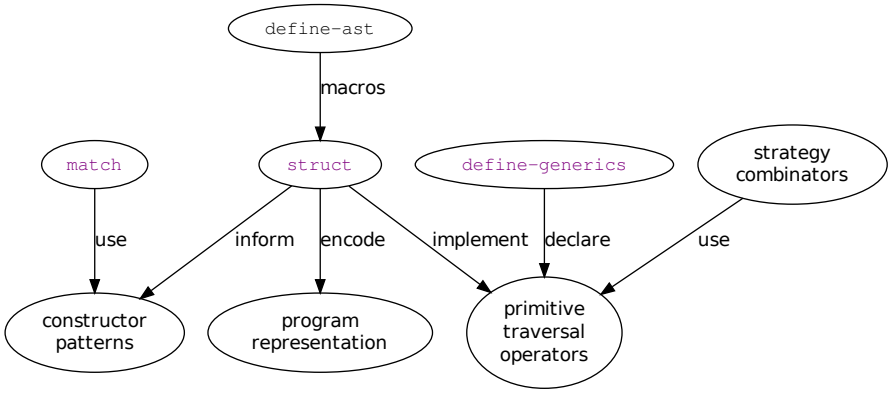


Figure 3.2: A Racket-based architecture for basic program transformation support. Structures provide data representation, and cooperate with **`match`** to support pattern matching. Macros generate structure-type-specific implementations of **`define-generics`**-declared generic methods to support composition of strategies that operate on concrete AST nodes.

3.3.1 Node Data Types

We use the term *node (data) type* (NDT) for types used to represent abstract syntax data in an AST. In Illusyn, such a type is declared using the ***`define-ast`*** macro, which expands to a structure type declaration of the same name, as well as some additional definitions. Field names must be specified, as one would for structure types. Additionally, one must use a keyword to indicate—for purposes of tree traversals—how many AST nodes the fields may contain: *`#:none`* for none, *`#:maybe`* for zero or one, *`#:just`* for exactly one, and *`#:many`* for any number. For example, we might declare the AST node types `BlockExpr`, `BlockStat`, and `LetStat` with

```
(define-ast BlockExpr (Ast Expr) ([#:none annos] [#:many ss]))
(define-ast BlockStat (Ast Stat) ([#:none annos] [#:many ss]))
(define-ast LetStat   (Ast Stat) ([#:none annos] [#:just def]
                                  [#:many ss]))
```

where `BlockExpr` gets one field `annos` (whose contents are not a part of the tree structure) and one field `ss` (containing a list of statements), while `(Ast Expr)` indicates that objects of this node type are subject to being accessed also through the operations of the `Ast` and `Expr` abstractions, whatever they may be.

3.3.2 Transformations

The information provided to **`define-ast`** is enough to generate type-specific supporting code required by Illusyn's Stratego-inspired ***`one`***, ***`some`***, and ***`all`*** primitive traversal combinators, which serve as a basis for composing term rewriting strategies to traverse and transform ASTs. Figure 3.2 provides an

overview of the resulting Illusyn architecture for the definition of transformations on concrete ASTs.

For an example of such a transformation, consider the following `ast-rm-Pass` function, for removing any "no-op" (i.e., `Pass`) statements from statement sequences. The tree traversal for `ast-rm-Pass` is defined in terms of a higher-level **`topdown`** strategy combinator, which in turn can be defined in terms of **`all`** and some other combinators. The rewrite function `rw` does the actual `Pass` removal, using Racket's higher-order **`filter`** function to transform the statement list `ss`. This is an example of a transformation that might be less fragile upon grammar changes if implemented in terms of a "statement container" abstraction:

```
(define ast-rm-Pass
  (topdown
   (λ (ast)
     (define (rw ss)
       (filter (negate Pass?) ss))
     (match ast
       [(BlockExpr a ss) (BlockExpr a (rw ss))]
       [(BlockStat a ss) (BlockStat a (rw ss))]
       [(LetStat a b ss) (LetStat a b (rw ss))]
       [_ ast]))))
```

3.4 Node Interfaces and View Data Types

In defining a node type in Illusyn with **`define-ast`**, one automatically gets at least one interface, consisting of operations reflecting the actual data representation of the type. Last section's `BlockExpr`, for example, gets the accessors `BlockExpr-annos` and `BlockExpr-ss`, and a predicate `BlockExpr?` for identifying values on which the accessors may be applied.

A **`define-ast`** declaration may also request the implementation of additional interfaces for the declared NDT; `BlockExpr`, for example, listed `Ast` and `Expr` as additional interfaces to be supported for that type. Any interface so listed must have been previously declared, with instructions for implementing it. Each such interface, when declared, gets a set of operations similar to an NDT's, as well as a dedicated "data representation." However, the NDT-like appearance of the interface is mere illusion in that no dedicated storage data type gets defined; instead, compatible NDTs are used for storage.

As Illusyn's node interfaces are effectively abstract data types (i.e., opaque, named "sorts" and associated operations) that define views (or projections) into the data of some underlying types, we use the term *view data type* (VDT) to refer to them.

While the functions of an NDT operate on known data types (with the exception of predicates, which accept any argument), Illusyn's VDT field access functions are single-dispatch generic functions, dispatching dynamically based on the type of the first argument. To support pattern matching, Illusyn implements exactly one view per VDT. A view implementation need not bypass the abstraction defined by the VDT interface; Illusyn implements each view as a syntactic transformation targeting VDT operations.

In Illusyn, one can declare a VDT using the ***define-view*** construct. Such a declaration must specify the fields of the abstract "record data type," and a way to map those fields to the fields of NDTs (or to compute their values based on NDT field values). That specification is then used to automatically add support for the VDT for any NDT whose declaration includes the name of the VDT (unless the specification is overridden for a specific NDT). For example, to define an `Assign` abstraction (further discussed in section 3.8) of something containing an `lv`alue and an `rv`alue, we can write the following, which merely states that the fields `lv` and `rv` should map to any NDT fields by name:

```
(define-view Assign ([#:field lv] [#:field rv]))
```

A VDT field may also `#:use` a different NDT field name in its mapping:

```
(define-view Assign2 ([#:field lv #:use x] [#:field rv #:use v]))
```

The original field specifications may be overridden in declaring an NDT, for any of the fields of any of its VDTs. Here, for example, we declare an `AssignExpr` NDT whose lvalue is exceptionally in a field named `x`. We therefore indicate that the abstract accessor `Assign-lv` should return the same value as the concrete accessor `AssignExpr-x`, provided that the receiver indeed is an instance of `AssignExpr`:

```
(define-ast AssignExpr ([Assign ([#:field lv #:use x])])
                       ([#:just x] [#:just rv]))
```

Currently, in Illusyn, mappings between VDTs and node types are specified separately for each abstract field (of a VDT), and there are two ways to specify them: (1) the *`#:field`* keyword indicates a field that is mapped by name; and (2) *`#:access`* indicates a field mapped using provided getter and setter functions.

For an example combining both kinds of field specifications, consider the following listing showing definitions of `Ast`, `Expr`ession, and `Lit`eral. `Ast` is defined to be a VDT for node types that contain a dictionary of annotations in a field named `annos`. The `Ast` abstraction is then used in defining an `Expr` VDT of typed expressions, whose type information is actually stored in their nodes' `'type` annotation rather than a field. One kind of `Expr` is a literal expression `Lit`, containing literal `data`:

```
; AST nodes have an annos field (with a non-AST value) for annotations
(define-view Ast ([#:field #:none annos]))

; Utility functions to get/set type from annos (indexed by symbol 'type)
(define (get-type ast)
  (hash-ref (Ast-annos ast) 'type #f)) ; default to a false value
(define (set-type ast t)
  (set-Ast-annos ast (hash-set (Ast-annos ast) 'type t)))

; Expressions have types, obtained from the Ast annotations
(define-view Expr ([#:access #:maybe type get-type set-type])
```

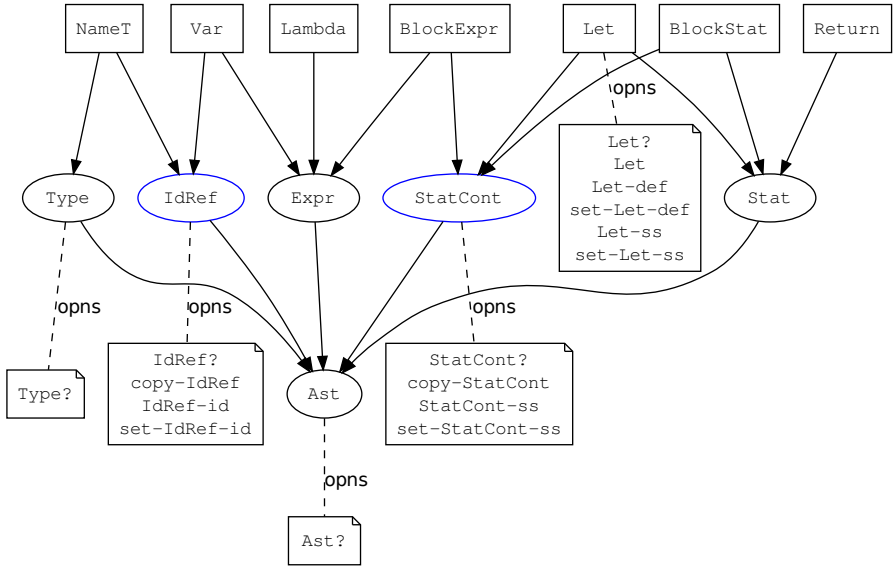


Figure 3.3: An abstract syntax hierarchy that captures not only grammatical relationships, but also common structure in node types. Rectangles denote NDTs, whereas ellipses denote VDTs. Only some of the operations (marked `opns`) associated with the types are shown.

```
  #:also (Ast))

; Literals have the Expr view
(define-ast Lit (Expr) ([#:none annos] [#:none dat]))
```

In the above, we give the `Expr` VDT a *field* containing the `type` of the expression (if known). `Expr`'s `#:access` to that information is defined in terms of the two functions `get-type` and `set-type`; the former retrieves the annotation from an `Ast` "node," whereas the latter sets it (with functional update). The functions are implemented in terms of Racket's **`hash`** operations, and the `Ast` VDT's `annos` field access (generic) functions.

The `Expr` definition's `#:also (Ast)` clause means that any NDT implementing `Expr` should also implement `Ast`. Illusyn's tracking of such dependencies makes it possible to abstract over VDTs in specifying them, by declaring that a VDT encompasses other VDTs. When generating code for an NDT, its VDT dependencies are collected transitively, and an implementation is emitted for all of them. As duplicates are ignored, and implementation specification overrides can only be given for the NDT, there is no ambiguity regarding implementations (thus avoiding the "diamond problem" associated with multiple inheritance).

Figure 3.3 illustrates the idea of having multiple interfaces to choose from per node type, on a case-by-case basis, according to which abstractions seem most convenient for a given transformation. One could also express a hi-

erarchy similar to the pictured one in terms of any host-language support for multiple inheritance, but in Illusyn any **`define-ast`**-defined node "hierarchy" is flat, with VDTs providing any subtyping-style abstraction, and macros providing code reuse by emitting VDT-related code into structure definitions.

3.4.1 Limits of the VDT Illusion

Since the goal is for VDTs to support commonality-capturing abstractions, and such abstractions necessarily only tend to include the "least common denominator" of encompassed concrete types, we naturally allow VDT-to-NDT mappings to be non-surjective. That is, a VDT may contain only a subset of the information of a concrete node. Thus, a VDT essentially defines a "sum type" of the relevant parts of the values of every one of the set of NDTs for which a mapping has been defined.

The illusion of VDTs being like concrete types necessarily breaks due to their possible partiality (of information), which means that there generally cannot be a constructor for a VDT. Instead, Illusyn provides what we call a *copy function* for VDTs (and also for NDTs, for consistency). A copy function is like a constructor in that it accepts (as arguments) the initial values of the fields of the object to be constructed, in declaration order. However, it also takes a "prototype" object serving two purposes: (1) it specifies the concrete type of the object to be constructed, and (2) it provides any data that the VDT does not concern. For example, if we defined a `StatCont` abstraction as

```
(define-view StatCont ([#:field ss]))
```

and listed it as supported for `BlockExpr` and the other statement-sequence-containing NDTs of section 3.3, then we might modify the `ast-rm-Pass` function to use the newly defined VDT's `copy-StatCont` function (with the prototype object `ast`) to incorporate the modified view contents into a new AST node. That is, if we **`match`** any statement container node, then we make a new one with a rewritten list of statements:

```
(match ast
  [(StatCont ss) (copy-StatCont ast (rw ss))]
  [_ ast])
```

As an exception to our earlier characterization of VDTs as "sums" of NDTs, Illusyn also supports another dimension of VDT partiality. For any VDT declared as *`#:partial`*[2], an implementing NDT may specify a *`#:predicate`* that dynamically determines which of the NDT values have mappings with respect to the VDT. This ability to guard VDT membership with a predicate adds expressiveness, for example allowing us to state that an *empty* sequence of statements is a non-operation:

```
(define-view NopStat () #:partial) ; non-operation
(define (nop? ast) (null? (SeqStat-ss ast)))
(define-ast SeqStat ([NopStat (#:predicate nop?)]) ([#:many ss]))
```

[2]We require the `#:partial` declaration, as this feature involves some overhead, due to a `#:partial` VDT's predicate itself having to dispatch to an NDT-specific predicate.

```
; Define DeVar match pattern
(define-match-expander DeVar
  (syntax-rules ()
    [(_ id t) (or (DeclVar id t)
                  (DefVar id t _))]))

; Use DeVar for matching
(match (DefVar 'x 'int (IntLit 5))
  [(DeVar id t)
   (list id t)])
; => '(x int)
```

Figure 3.4: Defining (with ***define-match-expander***) and using (within a **match** expression) a custom pattern matching form `DeVar` in Racket, for matching either `DeclVar` and `DefVar` objects and their common fields. The `id` and `t` identifiers within the **syntax-rules** expression are expected to bind patterns to be matched against field values. Here, the **or** form also constitutes a pattern, one that matches if either of its sub-patterns match.

3.5 Algebraic Views for Pattern Matching

Illusyn's AST nodes by their nature already have an algebraic term representation, due to Racket structures being used to represent them. Illusyn additionally exposes an algebraic "data representation" for each VDT "instance," to make it possible to pattern match either directly against the concrete data structure or any of its abstract representations. In essence, we get the ability to view one data structure as if it were another, in addition to being able to operate on it as if it were another.

We use the term *view* to refer to algebraic data representations realized through ADT operations (without knowledge of actual underlying structure). This appears to fit Wadler's original definition of the term, which states that "a view specifies how any arbitrary data type can be viewed as a free data type" [189]. In the Illusyn case, though, each view has a direct mapping to its corresponding VDT (as derived from the VDT name and the order of its declared fields), and it is only the mappings between abstract and concrete data types that must be specified by the program-transformation programmer.

As discussed in section 3.4, a VDT may contain only a subset of the information of a concrete node. Views are "complete," however, in that due to the direct view-to-VDT mapping, a view includes all of the data of its VDT; this is in line with Wadler's requirement of an isomorphism between "viewed" and "viewing" types for well-defined views [189].

The Racket pattern matcher has the necessary "hooks" to enable views to be implemented. Specifically, the set of patterns recognized by **match** can be extended by defining a *match expander* [168], which is essentially a macro that transforms patterns; the semantics of a new kind of pattern is defined by its translation to existing ones. Figure 3.4 shows an example of defining an expander and then **match**ing input against its pattern.

Illusyn's **define-view** macro generates a match expander to implement a view for the VDT being defined. As the view is implemented in terms of the abstraction provided by the VDT, it works for all concrete types associated with the VDT. For section 3.4's `Assign` VDT with the fields `lv` and `rv`, the generated view implementation would be roughly as shown below. For its patterns to match, the `Assign?` predicate must hold for the matched value, and the getters `Assign-lv` and `Assign-rv`, applied to the value, must produce

values matching the sub-patterns `lv` and `rv`, respectively:

```
(define-match-expander Assign
  (syntax-rules ()
    [(_ lv rv)
     (? Assign? (app Assign-lv lv) (app Assign-rv rv))]))
```

Matching against the pattern of a traditional view (i.e., as originally proposed by Wadler [189]) results in an implicit conversion from one data type to another [53]; in contrast, a VDT permits multiple possible concrete source data types for its view, and the "conversion" is more of a partial destructuring of one of the underlying data types, as defined by its specific getter implementations.

A drawback of the VDT abstraction is that the abstract constructs (e.g., `StatCont`) have no concrete syntax, thus precluding the use of concrete syntax in patterns and templates, as supported by some language workbenches (e.g., Rascal [111] and Spoofax [107]).

3.6 View-Directed Traversals

In most existing solutions with view-like "virtual representations," the abstraction extends to pattern matching, and often also construction, and in some cases congruence (e.g., Stratego's "overlays" cover all three). However, view-like abstraction mechanisms do not tend to support generic traversals over view data, or—more generally—enumerating the sub-parts of a view representation. This is rarely a problem, since in most solutions there is no data in the view data type that is not also reflected in the actual data type, meaning that a traversal over a concrete data structure will not miss any view data.

In contrast, a VDT may contain data that is not a part of the underlying term data structure, in the sense that enumerating a node's sub-terms with the concrete-structure-aware **`one`**, **`some`**, and **`all`** operators may not yield all the sub-terms of the VDT. This is because some VDT data could be stored in a `#:none` field, computed on demand (based on information in the node), or even stored entirely outside the node data structure (probably indexed by some information in each node).

The possibility of node types and VDTs having different tree structure (with different information content) introduces ambiguity to traversals. For example, the `Expr` VDT of section 3.4 stores its data in the `annos` fields of concrete nodes, whose AST node count is declared as `#:none`. When inspecting a node using the `Expr` view and operations, the `'type` annotation *looks* structural; however, a generic traversal such as the one shown in section 3.3's `ast-rm-Pass` would not visit such view "structure."

Some use cases may call for traversing a particular view or set of views instead of (or in addition to) traversing the concrete data structure directly, or even for carefully switching between views during traversals. For this reason, in Illusyn, the VDT abstraction extends from the local inspection-or-transformation contexts of AST traversals to the traversal-directing strategies themselves; the ambiguity cannot be resolved transparently, however, and we require explicit directing of traversals into views as desired.

Illusyn enables view-directed traversals by allowing any VDT to be declared as `#:traversable`. That causes generation of additional support code, which enables the use of ***make-view-one*** and similar macros to instantiate view-specific variants of the **one**, **some**, and **all** primitive strategy combinators. Any included higher-level combinators that use such primitives take optional arguments for overriding the default behavior, which is to use the (generic) **one**, **some**, and **all** operations to process traversable sub-objects. Illusyn's ***topdown*** strategy, for example, is implemented as

```
(define (topdown s [all all])
  (rec x (<* s (all x))))
```

where: the argument `s` is a sub-strategy to apply; the optional argument **all** has the top-level-bound, structure-traversing **all** strategy as its default value; ***rec*** binds a strategy for recursion (as in Stratego); and **<*** combines strategies sequentially (like `;` in Stratego).

As an example of a view-directed traversal, we might redefine the `Expr` VDT of section 3.4 to be `#:traversable`, allowing us to invoke **make-view-all** to get an `Expr`-specific variant of the **all** combinator. We might then use the resulting `Expr-all` combinator together with **all** to define a **topdown** tree rewrite `rw-tree`, traversing both concrete sub-terms *and* any type annotations in order to set all appearing type names to `'int`:

```
(define-view Expr ([#:access #:maybe type get-type set-type])
   #:also (Ast) #:traversable)
(define-ast NameT (Ast) ([#:none annos] [#:none id]))

(define Expr-all (make-view-all Expr))
(define (Expr+all s)
  (<* (when-rw Expr? (Expr-all s)) (all s)))

(define (rw ast)
  (if (NameT? ast) (set-NameT-id ast 'int) ast))
(define rw-tree (topdown rw Expr+all))
```

The **when-rw** combinator makes the specified strategy conditional on the specified predicate, reverting to the identity strategy when the predicate does not hold. Thus, `Expr+all` will first visit any `Expr` "view children" (where applicable), and then all the natural children.

3.7 Macro-Based Generation of APIs

The Illusyn implementation builds heavily on Racket's support for macros and generic interfaces. Macros are used firstly to implement the domain-specific sub-language (such as **define-ast** and **define-view**) to support declarative specification of AST "recipes," and secondly to do all the necessary code generation for AST API definition and implementation within the Racket language.

Generic interfaces (as declared with Racket's ***define-generics*** construct) enable the definition of operations that span different structure types, with dynamic dispatch to specific implementations; we use the term *generic method* for

operations of generic interfaces.[3] NDT-specific implementations of printing, equality comparison (with regards to Racket's **equal?** relation), serialization, and strategic term rewriting support, for example, are all macro generated by Illusyn, and defined as generic methods of the specific structure type. Each VDT also gets its own generic interface, and node-type-specific VDT support code can thus be treated similarly.

For code generation, Illusyn uses Racket's built-in facilities for Racket-code manipulation, in an idiomatic way. For an idea of what such code-generating code looks like, consider the function `make-view-pattern` below, which is a simplified version of the one that the **define-view** macro invokes internally to generate match expanders such as the `Assign` one in section 3.5. The function takes a view identifier and list of field identifiers, and returns syntax for defining the corresponding match expander:

```
(define-for-syntax (make-view-pattern view-id fld-ids)
  (define name (syntax-e view-id))
  (with-syntax ([pat-name view-id]
                [view? (format-id view-id "~a?" name)]
                [(get ...) (for/list ([id fld-ids])
                             (format-id view-id "~a-~a"
                                        name (syntax-e id)))]
                [(pat ...) (generate-temporaries fld-ids)])
    #'(define-match-expander pat-name
        (syntax-rules ()
          [(_ pat ...)
           (? view? (app get pat) ...)]))))
```

Macros provide a full-fledged interface to the Racket language implementation, and indeed large parts of Racket itself have been implemented as macros. This means that it is possible for custom, macro-defined language to have notation that is indistinguishable from "native." Racket macros also offer "proper" abstraction over syntax, in that they respect lexical scope (by default); i.e., they preserve the meaning of variable bindings and references during macro expansion [52].

A key mechanism for macro cooperation in Racket is its support for *general compile-time bindings*, i.e., binding of identifiers to arbitrary compile-time information [73]. The mechanism is particularly convenient for maintaining scoped, hygienic "name tables" at macro-expansion time, and Illusyn relies solely on it for its cross-macro communication of information about VDTs.[4] Specifically, the **define-view** macro stores information about the VDT being defined, and binds it by name in order to make it available to the **define-ast** macro; the latter can then look up the information in order to implement the VDT mappings for any node type that lists the VDT as supported.

Other implementations of views have previously been described. Okasaki, for instance, has proposed a way to add views into Standard ML, defining the

[3] For additional implementation flexibility, Illusyn does not always define generics in terms of **define-generics**; instead, it in some cases directly uses the underlying Racket facility of "structure type properties" to implement generics, by storing a type-specific function table into a property. We speak of generic interfaces and methods also in these cases.

[4] Macro systems without general compile-time bindings might instead explicitly maintain a (mutable) name table in their macro-expansion-time state.

proposed semantics via a source-to-source translation into "plain" Standard ML [134]. As Standard ML lacks an extension mechanism, implementing Okasaki's solution would involve modifying the language implementation, or writing an external source-to-source compiler. Racket's macros have provided us with a convenient way to embed our solution within the language, with a result that is integrated and compatible with existing Racket tools.

As demonstrated earlier, e.g., by Kiama [158] for Scala, it can take relatively little code to create a capable language processing toolkit based on domain-specific language embedding into a general-purpose host language. In the Illusyn case it is macros that are the enabler for embedding, and the full-featured host language complements the capabilities of the library itself. For example: while Illusyn provides an AST data structure, Racket must provide any others, such as lists, sets, and dictionaries; Racket's dynamically scoped variables appear to be particularly useful for tracking a transformed language's scope; and Racket has sufficient features and libraries for implementing an entire compiler, and its integration with other applications.

3.8 AST Abstraction Scheme

Macros can be used to define syntax for DSL constructs for specifying desired AST node data content, which we assume is specified by listing named data fields, possibly with some attributes. There can also be syntax for specifying commonalities, which might entail having the same data fields, or the same data content after some extracting, combining, or translation.

With those things specified in a DSL, macros can also be used to generate the implementations of both the node data structures and their commonality-capturing abstractions. Efficient implementations thereof may require intricate language-dependent code, which can hopefully be generated with macros, and which the language can hopefully support in terms of run-time features. At least we can hide such details under simple interfaces, in order to enable client code to be written in relatively primitive and portable language. We can make the interfaces uniform in style to also encourage building of further abstractions on top; any macros we write will know what kind of operations and naming to assume.

Our AST design builds on the idea that almost no matter what kind of type definitions we are generating for purposes of AST data representation, we can hide them within one or more abstract data types. An *abstract data type* (ADT) has a public name, a hidden representation, and operations to create, combine, and observe values of the abstraction [38]. At the interface level, adding a new node or abstraction merely involves adding a collection of operations for it.

The operations can be generated as ordinary functions, and any representational types (within an abstract type) might be record types or classes or type-name-tagged tuples with the requested fields. For example, for the AST node type `Var` with the field `id`, we would at least want operations like a constructor `make_Var`, a predicate `is_Var`, and a field accessor `Var_id`, with `is_Var` to be used for guarding access to `Var`'s data. For C++, for example, we might generate

```
struct Var : AstData { // AstData is polymorphic
  explicit Var(std::string const& id) : m_id(id) {}
```

```
  std::string m_id; };
// Ast is an alias for std::shared_ptr<AstData>
Ast make_Var(std::string const& id) {
  return std::make_shared<Var>(id); }
bool is_Var(Ast const& ast) {
  return typeid(*ast) == typeid(Var); }
std::string Var_id(Ast const& ast) {
  return dynamic_cast<Var&>(*ast).m_id; }
```

In a statically typed language our collection of AST node types could all be given the same static type (e.g., `Ast`, above), or alternatively some groups of node types could be associated with different ADTs to impose type constraints at the interface level. We consider all node types as technically unrelated, and establish any relationships by generating (for an ADT) additional generic operations that span multiple (or no) node types, provide access to certain (common) portions of available information, and together constitute an interface similar to actual nodes' interfaces.

For example, we might abstract over any node types representing assignment operations (e.g., `AssignStat`) by generating `Assign` operations capable of accessing any such object type's lvalue and rvalue expressions. Such operations require dynamic dispatch to a specific type's operations, as we want to support an open-ended set of `Assign`-implementing types. We can achieve that with a dispatch table of some kind, and in C++ we can get a language-provided "vtable" by making `Assign` an abstract class of pure virtual functions:

```
struct Assign {
  virtual Ast Assign_lv() = 0;
  virtual Ast Assign_rv() = 0;
};
struct AssignStat : AstData, Assign {
  explicit AssignStat(Ast const& lv, Ast const& rv) :
    m_lv(lv), m_rv(rv) {}
  Ast Assign_lv() { return m_lv; }
  Ast Assign_rv() { return m_rv; }
  Ast m_lv, m_rv; };
// definitions for make_AssignStat, is_AssignStat,
// AssignStat_lv, and AssignStat_rv omitted
// (they are similar to Var operations)
bool is_Assign(Ast const& ast) {
  return dynamic_cast<Assign*>(ast.get()) != nullptr; }
Ast Assign_lv(Ast const& ast) {
  return dynamic_cast<Assign&>(*ast).Assign_lv(); }
Ast Assign_rv(Ast const& ast) {
  return dynamic_cast<Assign&>(*ast).Assign_rv(); }
```

Where the host language does not support incremental definitions, macros can collect information about abstractions into expansion-time state, and then emit full definitions at once. For example, C++ does not have open classes, but we can list `Assign` as a superclass for `AssignStat` if we have seen `Assign`'s specification before `AssignStat`'s.

While there are host-language-influenced implementation choices to make in realizing abstractions such as `Assign`, altering those choices can be cheap if the implementations are generated from declarative specifications. The challenge is in devising ways to specify the implementations, and preferably once per abstraction (not once per concerned node type). In Illusyn, we support name-based mapping of abstract-to-concrete fields, for example, and that would be sufficient for the `lv` and `rv` fields of `Assign` and `AssignStat` here; more flexible ways of specifying relationships could be devised.

Having arranged for the generation of uniformly named operations for both objects and abstractions, one can start devising other abstraction-friendly facilities (syntax or operations) expressed in terms of those operations. Two prominent examples of such facilities in Illusyn are generic tree traversals and pattern matching.

Existing AST generators tend to support abstraction over specific AST structure in the form of generic traversals of some kind. Stratego [28], for example, provides primitive traversal operators (**`one`**, **`some`**, and **`all`** generic functions) for all abstract syntax tree node types. We can also express such traversal primitives in terms of our data access functions, whether for a node or an abstraction. Traversals over actual or abstract structures can thus remain fairly general until data of interest to analyze or transform is encountered.

Analyses and transformations within ASTs are often performed by pattern matching to identify a particular subtree, and then replacing the subtree with an information-enriched or transformed one. Programming against ADTs need not mean the loss of pattern matching.

To support pattern matching—with uniform treatment of nodes and abstractions—we can generate macro definitions that translate match patterns into invocations of data access operations. As an example, consider the following function `fm`, given in Racket syntax (due to the lack of pattern matching syntax in C++); the function uses a pattern **`match`** expression to record the identifier `id` of any argument variable that appears on the left-hand side of an assignment:

```
(define (fm ast) ; (-> Ast? void?)
  (match ast ; against the following clauses
    [(Assign (Var (? arg? id)) _) (add-mut-arg! id)]
    [_ (void)]))
```

In `fm`, the recording is done by calling `add-mut-arg!`, for L-value `Variable` identifiers indicated by the predicate `arg?` (the `(? ....)` pattern matches according to the specified predicate, whereas the `_` pattern matches anything). The above `(Assign ....)` pattern refers to the `Assign` abstraction, whereas the `(Var ....)` pattern refers to the concrete object type `Var`, but both patterns translate in a uniform way. A C++ translation of `fm` (in terms of `Var` and `Assign` operations given earlier) might for example be

```
MGL_FUNC void fm( Ast const& ast ) {
  Ast temp23;
  std::string temp29;
  is_Assign(ast) ?
    ((temp23 = Assign_lv(ast)),
     (is_Var(temp23) ?
```

```
      ((temp29 = Var_id(temp23)),
       (is_arg(temp29) ? add_mut_arg(temp29) : fm_f16())) :
      fm_f16())) :
    fm_f16();
}

MGL_FUNC void fm_f16(  ) { }
```

Even Magnolisp—a small research language implemented with Illusyn—supports first-order functions and conditional expressions. Thus, while its language is not rich enough to host Illusyn's entire feature set, it is capable of *using* AST APIs defined according to the scheme described here; in fact, the above C++ translation of `fm` was produced by the Magnolisp source-to-source compiler.

3.8.1 Host Language Extensibility Requirements

Implementing the described scheme requires a macro system of moderate sophistication. In particular, there must be support for programmable macros with access to macro-expansion-time state, and it must be possible to generate top-level definitions. This rules out a large portion of languages with macro systems, as many of them only support expression generation; still, at least the *Honu* [147] language has the required support, as do many flavors of Lisp, and easier-to-implement models of macro expansion of the required power are being discovered [68]. Preprocessor-based implementations of the required macro facilities are also possible, as demonstrated by *sweet.js* [48] for JavaScript.

For best results, the host language should also have a general, extensible pattern matcher (one that also supports *non*-AST data types). There are languages (like F# and Scala) featuring integrated, extensible pattern matching. Racket's extensible pattern matcher, in turn, has a macro-based implementation [168], which is an option for languages that otherwise lack a suitable matcher.

For implementing views as macros, having language support for macro-generating macros allows for more implementation options, but is not a requirement. In discussing match pattern translation above, we used the term "macro" loosely, to mean a syntax transformer function with the same signature as macros have; a pattern transformer function need not be *bound* as a macro, if there are other means for looking it up for pattern translation. Allowing macro-binding macros can pose parsing challenges in languages without a uniform syntax (e.g., formed out of S-expressions) [2; 147].

3.9 Related Work

Racket's **`match`** [168] cooperates with **`struct`**ure type definitions in such a way that views are already supported for structures. In pattern matching against a structure, the data representation of one of its supertypes may also be used, even if the supertype is semantically abstract. For example, we may use an `Expr` pattern to match against a `Var` value, where `Expr` is an abstract supertype without an exposed constructor:

```
(define-syntax-rule (abstract-struct n more ...)
  (struct n more ... #:constructor-name ctor))
(abstract-struct Expr (t))
(struct Var Expr (id))

(match (Var 'int 'x)
  [(Expr t) t]) ; => 'int
```

Illusyn's AST node type relationships are not expressed in terms of **struct**ure supertype declarations, leaving more flexibility with regard to what relationships are expressible.

In Wadler's views mechanism [189], each **view** has a "data type" with one or more algebraic variants, and a bi-directional correspondence to another data type; the conversions **in** and **out** of the data type are specified similarly to functions. The bi-directionality enables a view's algebraic variants to appear both in patterns and expressions. In our case there's exactly one variant per VDT, and it can appear in patterns, but has no constructor; instead, there is a copy function for prototype-based construction.

Wadler suggests implementation either by: (1) having a real data type, and inserting calls to conversion functions in the appropriate places; or (2) avoiding the data type, instead translating view pattern and constructor appearances into expressions concerning the "viewed" type (with patterns translated into invocations of a higher-order `viewcase` function). The latter approach resembles ours in that a view's data type is mere illusion, and its data access is actually to an underlying data type. In our case, though, a VDT can span multiple underlying data types in an open-ended way, meaning that we cannot translate away dynamic dispatch to type-specific operations.

Okasaki has proposed a views extension for Standard ML, defining its semantics via source-to-source translation [134]; our implementation is based on macro expansion, which is also a form of source-to-source translation. Okasaki's views consist of "viewtypes" and one-way "view transformations." View representations can only be used for pattern matching, not for construction; view transformations are implicitly invoked to convert from the concrete type to the view type, for matching against view patterns. Okasaki's proposed translation involves view-specific ML **datatype**s; in our solution underlying data types are used, and it is operations that must be generated for a VDT.

Stratego's "overlays" [179] are also integrated into a program transformation toolkit. They are like Wadler's views in that an overlay is an abstract algebraic variant which is isomorphic with its underlying representation (of real algebraic variants). Overlays are defined as equations between representations (possibly involving other overlays), and the mappings are not necessarily total; data not reflected in an overlay must map to constant values of the underlying representation, while with VDTs such missing pieces of data are copied from a prototype object. One motivation for the Stratego feature is to enable overlaying a language on top of a more generic representation language, while VDTs aim to make it convenient to have multiple similar "representations" for the types within a single AST hierarchy.

Tom [128] can be used to define a term representation for existing ASTs, for purposes of pattern matching; the same facility could be used for declaring multiple representations for the same underlying type. Tom is a tool for adding

a pattern matching sub-language to an existing language, implemented as a restricted source-to-source compiler that only partially parses the host language (for easier adaptation for multiple hosts). The sub-language includes a **`%match`** construct and declarations for term types and variants. Racket's **`match`** and **`define-match-expander`** form a similar sub-language, implemented in terms of a general-purpose compile-time syntax transformation system (i.e., macros) rather than a dedicated compiler.

Scala supports pattern matching on both concrete data types (with "case classes") and abstract data types (with "extractors") [53]. These Scala facilities are comparable to Racket's built-in support for **`match`**ing against **`struct`**ures and defining match expanders, respectively. A major difference is that an extractor defines a run-time data conversion, whereas a match expander defines a compile-time pattern transformation. An extractor is suitable for VDT match pattern implementation in that its `unapply` method (as applied in patterns) permits multiple source data types (e.g., based on subtyping or overloading).

F# has "active patterns" [165] as a pattern matching abstraction, one that is similar to extractors. F# includes dedicated syntax for "structured names" of patterns, which may be defined separately from data types. A structured name definition requires an associated function expression to specify how to convert "viewed" data to "viewing" data.

JMatch [121] extends Java with pattern matching support. Its **`interface`**s may declare "pattern methods," allowing data-type-specific implementations of named patterns; this is more flexibility than is required by our VDT implementation, where each view pattern has a single implementation, but where the expanded pattern may include calls to generic methods of an interface. JMatch patterns also support iteration (over a set of matches), which can be convenient in implementing tree traversals, for example. JMatch should be a good basis for further language extensions to support declarative AST implementation.

The functional programming concept of "lenses" is related to VDTs in that a lens can also provide a view into partial data (of some larger structure). A difference is that a VDT is a type, whereas a lens is a value. For any field of a VDT, a lens can be instantiated in terms of the field's (purely functional) accessors. Lenses might complement traversals by providing a composition-friendly way to access and modify AST sub-parts in a local context, without explicit de- or re-construction of nodes.

Some of the domain-oriented languages (like Rascal and Stratego) have features like generic traversals and pattern matching built-in, allowing ASTs to be expressed concisely, without additional code generation being required. However, to date these tools have not provided much support for exposing multiple interfaces per node type, or traversals guided by interfaces of choice. Stratego's overlays are one exception; with them, alternative representations can be exposed, and traversing the underlying representation also results in traversing any overlays, making choice of interfaces largely unnecessary.

Kiama explores the advantages and constraints of embedding a program transformation toolkit into the Scala language [159]. It makes heavy use of the specific facilities of its host language, including features such as case classes, pattern matching, implicit conversions, extractors, **`lazy`** values, call-by-name parameters, and operator notation. By contrast, our aim has been to devise a design that requires no exotic features from the run-time language, instead

shifting responsibilities onto the macro system.

Kiama's embedding approach does not enable static correctness checking beyond that which is provided by the host language; Racket embeddings, on the other hand, have no host language type system to exploit, but domain-specific static analysis is possible during macro expansion [169]. Illusyn's static semantics enforcement is limited to basic checks of consistency between NDT and VDT declarations.

3.10 Discussion

The Magnolisp compiler has so far been the only application for Illusyn, although we plan to adopt the approach (in some form) in our experimental compiler for the general-purpose Magnolia programming language, which will be a more realistic test of its usefulness. While there have been challenges relating to infrastructure, architecture, and algorithms in implementing Magnolisp, writing the required transformations with Illusyn has not in itself been difficult. The major uses of Illusyn in Magnolisp are: construction of nodes during parsing in the front end; analysis and language-independent transformations in the middle end; and C++ translation and pretty printing in the back end.

3.10.1 Technology Choices

Racket is a modern functional language, and we have found it to be a suitable and complementary host for our domain-specific program transformation sub-language. In programming program transformations, we have felt empowered by the combination of Illusyn and particularly its host language's module system and pattern matching and macro facilities.

In developing Illusyn's library-based DSL, we have not felt that we have needed to make language design tradeoffs, compared to what would have been possible had we instead created a special-purpose external DSL for the domain; we do not know if this is due to our lack of imagination or design sense, however.

As discussed in section 3.8.1, there presently are few languages beyond Lisps which have the macro capabilities required by our AST generation approach, but it might nonetheless have applications beyond those languages. For instance, while we have only discussed macro-based implementations, more generally extensible languages such as mbeddr [187] and SugarJ [55] should easily be able to host a variant of our scheme. We also see macro system reuse as a feasible option for newly designed languages to incorporate a sufficient extension facility, with more reuse potential than when inventing a custom macro system.

We have designed our AST generation scheme to require little from the target core language (to which macros must ultimately expand). We did this as we believe macro systems (and their macro languages) have the potential to serve as cross-language platforms for reusable language features. A macro system can serve as such a platform if:

- The macro system supports integration with other languages. The language-independent syntactic-macro processor Marco [117] is one ex-

ample of such a system, and Racket's macro system can also be adopted for other languages, with some constraints [88].

- The language of the macro system is used for macro programming, even though it may be very different from the target language. The Terra [47] language, for example, incorporates a separate language (i.e., Lua) for its meta-programming, while Magnolisp uses Racket as its macro programming language.
- There is a cross-language subset of core language that can be targeted in translating reusable constructs. Magnolisp, for example, shares some core language with Racket, which allowed the `fm` function of section 3.8 to reuse Racket's **`match`** macro as is.

3.10.2 Usability and Abstraction Power

Our scheme for extragrammatical abstraction in ASTs has proven to be workable: as described in section 3.4 in particular, we have implemented a number of ways to declare abstractions for nodes, and the design has readily accommodated incremental implementation of new declaration syntax. Most often, such additions have lead merely to generating additional functions, which is also unlikely to break existing client code. We recall only one instance where we have ended up with a naming conflict with newly generated code: we started generating equality relations for NDTs, but we already had a manually implemented `Id=?` operation (of a different semantics) for our `Id` NDT.

VDTs are our sole mechanism for declaratively expressing relationships between AST node types. VDTs subsume subtyping with respect to what type hierarchies can be defined. We have used them extensively for expressing grammar-derived relationships, in order to process `Expr`essions and `Def`initions generally, for example.

We have only occasionally used VDTs for expressing non-grammatical relationships; usually, when we have, the motivation to define a VDT has arisen from the desire to avoid duplication of similar case clauses in **`match`** expressions and other conditionals. One example of such case clause avoidance was the `StatCont` use of section 3.4.

A common VDT use case within Magnolisp is to abstract over the distinction between expressions and statements, for abstract syntax that has both variants. As an example, consider the VDT `If`, which abstracts over "two-armed" conditionals, whether statements or expressions. While the distinction between statements and expressions matters e.g. during type inference (since expressions are typed), it can be ignored when optimizing away constant condition checks. The definition of `If` and a function `rw` performing the optimization, extracted from a version of the Magnolisp compiler, are as follows:

```
(define-view If ([#:field c] [#:field t] [#:field e]))
(define-ast IfExpr (Ast Expr If)
  ([#:none annos] [#:just c] [#:just t] [#:just e]))
(define-ast IfStat (Ast Stat If)
  ([#:none annos] [#:just c] [#:just t] [#:just e]))

(define rw
```

```
(bottomup
 (λ (ast)
   (match ast
     [(If c t e) (cond
                  [(TRUE? c) t]
                  [(FALSE? c) e]
                  [else ast])]
     [_ ast])))
```

From a user's point of view, our DSL syntax is not especially easy to remember, but the underlying scheme is conceptually simple: pick your storage representations first (with **define-ast**), focusing on what information must be stored; then, almost orthogonally, pick additional data types to expose (with **define-view**), focusing on what transformation tasks will be required. From a DSL implementor's point of view, the scheme is similarly simple in concept: treat both NDTs and VDTs as abstract data types, and generate a uniform (to a possible extent) set of operations and a single view for each.

A complete illusion of node types and VDTs being alike cannot be attained. For one thing, some views are even conceptually abstract, and thus cannot have a constructor (e.g., consider a non-terminal such as `Expr` that does not exist concretely). We instead generate copy functions for concrete nodes as well, to make it possible to lessen the client code impact of a node type being turned into a VDT; this still does not complete the illusion, however, as something must ultimately be constructed for prototype-based instantiation to be possible. Another aspect of broken illusion is traversals, where there is fundamental ambiguity in choosing how to traverse into a multi-interface node.

Our attempt at uniformity between concrete and abstract nodes does at least result in better API stability than with Racket's structure subtyping. In the following scenario, for example, the first `B` will have getters `B-a` and `B-b`. If we split the `a` field into a supertype `A`, as in the second definition, then `B`'s getter names will change to `A-a` and `B-b`. In the third definition, **define-ast** generates getters `B-a` and `B-b` for `B`, and changes in `B`'s `A` membership do not change `B`'s own operations; VDT operations are separate, and in this case `A`'s sole getter will be `A-a`:

```
(struct B1 (a b))
```

```
(struct A (a))
(struct B2 A (b))
```

```
(define-view A ([#:field a]))
(define-ast B3 (A) ([#:none a] [#:none b]))
```

Our VDT definition support is intended to make it straightforward to specify common-case abstractions over sets of AST node types, and `#:field` and `#:access` field definitions have covered typical use cases in Magnolisp. We believe that our general design could accommodate other kinds of field definitions. For example, one might want an `#:access` field definition to alternatively be specifiable as syntax-transforming compile-time functions that are parametric in information about the concrete structure type being accessed,

allowing for some variation between concrete implementations, for a single specification.

The idea of generating interfaces as collections of standalone (possibly generic) functions seems to be a good match for Stratego-style rewriting with higher-order functions.

One way in which Illusyn differs from Stratego is that lists (or cons cells, rather) are not considered as terms for purposes of traversals. Consequently, a bottom-up traversal does *not* traverse lists in a reverse order, as might be useful for certain control-flow sensitive rewrites, for example. Illusyn has special-case behavior for traversing `#:many` fields, and list-type field values are assumed; this also means that there is no support for fields containing sets or dictionaries of terms, for example. We will revisit this design if necessary, but one of its advantages is the availability of some static knowledge about field contents, allowing direct use of Racket's optimized list operations as appropriate.

3.10.3 Efficiency

Our abstraction scheme should generally permit a reasonably efficient implementation, comparable to inheritance-based subtyping: no dynamic dispatch is required for NDT operations; one level of single-argument dynamic dispatch is required for VDT operations, as VDTs represent open-ended sets of types[5]; view data type operations manipulate underlying data types directly, thus requiring no data conversions; and, view patterns are translated away statically, into concise code with ADT-operation-based data access.

The efficiency of dynamic dispatch, of course, depends on the way it is realized in terms of the target language. Some languages optimize such dispatch, for instance by maintaining "lookup caches" of recently looked-up type-specific routines, or even "inline caches" [46; 98] of lookup results directly at call sites; where such language support is available, macros are hopefully able to generate code to exploit it.

In Illusyn, the run-time cost of individual VDT field accesses depends on user-provided specifications. An `#:access` field access performs arbitrary computation by definition. However, at least in Magnolisp, `#:field` access is the common case, with consistent overhead. VDT getters are particularly interesting in that they can appear multiple times in a VDT pattern translation (as shown in section 3.5's `Assign` example). Due to the getters' type-specific implementation, fetching the value of a `#:field` involves one generic method dispatch (which effectively also checks for VDT membership), and an **`unsafe-struct*-ref`**[6] of the appropriate concrete field. In comparison, as of Racket 6.3, a regular structure field access within a **`match`** clause entails only a (more general and potentially slower) **`unsafe-struct-ref`** at a statically known offset.

[5] In contrast, if there was exactly one concrete type per VDT, then it would be possible to make the abstraction zero-overhead, in the sense that macros could replace all VDT accesses with accesses of the underlying type. Similarly, handling multiple *known* concrete target types might also be possible by translating into multiple case clauses, but at the cost of a corresponding code size increase.

[6] Our generated code uses unsafe, type-check-omitting operations where the specific type has already been established due to generic method dispatch.

One place where the cost of internal abstraction over data types shows is that the VDT copy functions inferred by Illusyn for certain kinds of field specifications (i.e., where there are both `#:field` and `#:access` fields, or multiple `#:access` fields) may be of sub-optimal efficiency, involving multiple calls to Racket's **`struct-copy`** function where one should suffice. This could perhaps be addressed with more creative `#:access` specifications. Barring such innovations, one may—at some cost to maintainability—resort to the supported option of specifying node-type-specific (and optimized) implementations of copy functions. The **`define-ast`** form allows for such overrides for each listed VDT with an optional `#:copy` clause.

For purposes of comparing alternative implementations of field access, we have done basic performance measurements. In case (a), we access a field shared by Racket structure types via their base type; this is our baseline case, as the offset of the data within all derived structures is known statically. In case (b), we use unrelated types, and a **`cond`**itional to enumerate all the alternatives. In the (c) case, we fetch via a VDT-defined `#:field` accessor, and in the (d) case, we fetch via a field accessor mixed-in as a Racket-object-system `trait`. The (a) case is not comparable to (c) and (d) in the sense that only the latter support "multiple inheritance." The reported times are in milliseconds, for fetching the value of a field from 1000 objects, and doing that 10000 times. For instantiating the objects, we select from either 1, 3, 5, 10, 30, or 50 different types of objects, which is significant in the (b) case:

	1	**3**	**5**	**10**	**30**	**50**
(a) via base **`struct`**	780	780	784	784	784	780
(b) **`cond`** cases	792	952	952	1012	1240	1452
(c) via VDT	1108	1156	1172	1204	1208	1236
(d) via **`class`** trait	2108	2112	2112	2116	2152	2172

We observe from the results that: unless the number of relevant concrete types is unusually large, performance is not a reason to favor VDT access instead of manually enumerating specific types; and that the overhead of accessing a **`class`** instance field through a method call appears significantly greater than that of VDT generic method dispatch.

Another aspect of efficiency to consider is code size. In comparison to plain Racket structure types, Illusyn's NDTs and VDTs get more operations generated for them. It is likely that a large portion of the operations do not get used, but this is also the case with regular structures.

The increase in the amount of code generation is detrimental in at least two ways: macro expansion takes time; and code footprint may increase. The first issue can be alleviated by avoiding repeated macro expansion; Racket includes support for managing modules in a byte-compiled form in which macro uses have been expanded. Some languages may drop unused operations (e.g., Magnolisp does), thus avoiding the footprint increase. Racket—to our knowledge, as of version 6.3—offers no such guarantees, and we try to avoid code bloat by making the generation of less commonly required operations optional; for example, we assume that most VDTs will not be traversed, and require the `#:traversable` attribute for VDTs that should get the traversal operations.

3.11 Conclusion

There are programming languages (e.g., Scala) that feature support for interfaces and reusable implementations, and should thus enable convenient and concise manual implementation of abstract syntax trees with extensive use of interface-based abstraction. However, not all languages have such features, and there has so far been little support for non-OO interface-based abstraction for AST APIs.

We have presented a scheme for providing AST APIs based on macros, such that there is support for declaring abstractions over collections of AST node types. This essentially means that it is possible to define abstract interfaces for nodes, even when the target language has no concept of interfaces (independent of implementations). There are also some advantages over what one could achieve in terms of traditional object-oriented language features: each interface gets syntax for pattern matching; interface implementations need not be hand written, but are provided from declarative specifications; and consequently, it is feasible to provide type-specific, optimized implementations of commonly used operations; and similarly, it is possible to radically modify implementations without modifying client source code.

The language that we have so far devised for declaring abstractions is not especially powerful, but it is sufficient for expressing both grammatical relationships (with terminals "inheriting" fields from associated non-terminals) and purely structural commonalities.

One of our more unusual goals has been to blur the distinction between concrete and abstract AST nodes, in an attempt to make the choice between concrete and abstract types less crucial. Towards this end we have adopted the idea of views to provide a representation for abstract types. In most existing view-utilizing solutions each view corresponds to a specific data type, whereas in our solution a view corresponds to an interface, and may therefore represent multiple different concrete types. Consequently, traversals over interface-implementing nodes are ambiguous, and we allow such ambiguities to be resolved through parameterization of traversals with sub-term-enumerating operations, while defaulting to traversing over concrete structure.

We have described an implementation of these ideas in the form of a Racket library named Illusyn. The library contains a collection of higher-order functions enabling concise, composition-supporting implementation of traversals and rewrites over ASTs; the included functions are designed to interact with less-general ones that are generated from declarative specifications by the library's macros.

Our heavily macro-based implementation scheme requires little from the run-time language, apart from (preferably user-definable) composite data types and some support for implementing dynamic dispatch based on concrete type; in particular, there need not be support for multiple (or even single) inheritance, or more advanced constructs such as traits for expressing and mixing in common code across node types. This suggests that other languages with a suitable macro system could also similarly implement an AST support library within the language. A preprocessor-based macro system (e.g., sweet.js) might enable implementation for languages without built-in macros.

The ideal implementation language would also have a general-purpose view-like mechanism (e.g., "extractors" in Scala [53], "active patterns" in

F# [165], or "pattern methods" in JMatch [121]), or other suitable hooks into its pattern matching facility. Otherwise, one can resort to creating a separate, macro-based pattern matcher with the required hooks [168].

Acknowledgements

This research has been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language. Vadim Zaytsev, May-Lill Bagge, and anonymous referees have provided useful comments during the writing process.

CHAPTER 4

Permission Management

In this dissertation, I use failure management as an example language technology use case. It is a particularly relevant concern for always-on, resource-constrained embedded devices, intended to run for long periods of time without supervision.

Mobile platforms have popularized security models based on fine-grained permissions, making the lack of required permissions an additional potential cause of failures affecting niche platforms. Permissions are more of an issue of error prevention than handling, as permissions can in a typical case be assumed granted for a running application, *if* they have been requested. Getting permission requests right, in turn, is a matter of having the necessary (static) information about product configurations, and language support can be helpful here, too.

In this chapter, we present a language-supported solution for permission management, intended to automate the required-permission-bookkeeping aspect of product configuration management. The idea is to infer the permission demands of a given software product based on the APIs it uses, rather than to declare the requirements separately for each product. The inferred information can then be fed to the build system as a permission request declaration of the appropriate format.

The solution is cross-platform, and it assumes that products are composed in a static-reasoning-friendly programming language. It furthermore assumes that product compositions are static, and that any (opaque) foreign-language implementations of operations are annotated with their required permissions.

I presented this paper at NordSec 2013 in Ilulissat [Hasu et al., 2013].

Inferring Required Permissions for Statically Composed Programs

Tero Hasu Anya Helene Bagge

Magne Haveraaen

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway

Abstract

Permission-based security models are common in smartphone operating systems. Such models implement access control for sensitive APIs, introducing an additional concern for application developers. It is important for the correct set of permissions to be declared for an application, as too small a set is likely to result in runtime errors, whereas too large a set may needlessly worry users. Unfortunately, not all platform vendors provide tools support to assist in determining the set of permissions that an application requires.

We present a language-based solution for permission management. It entails the specification of permission information within a collection of source code, and allows for the inference of permission requirements for a chosen program composition. Our implementation is based on Magnolia, a programming language demonstrating characteristics that are favorable for this use case. A language with a suitable component system supports permission management also in a cross-platform codebase, allowing abstraction over different platform-specific implementations and concrete permission requirements. When the language also requires any "wiring" of components to be known at compile time, and otherwise makes design tradeoffs that favor ease of static analysis, then accurate inference of permission requirements becomes possible.

4.1 Introduction

Permission-based security models have become commonplace in real-world, consumer-faced operating systems. Such models have been adopted mostly

This is a preprint of: Hasu, T., Bagge, A. H., and Haveraaen, M. Inferring Required Permissions for Statically Composed Programs. In Proceedings of the 18th Nordic Conference on Secure IT Systems (Ilulissat, Greenland, 18–21 October 2013). NordSec 2013. Lecture Notes in Computer Science *volume 8208, 2013, pp 51–66.*

for mobile OS platform security architectures, partly because smartphones are high-utility personal devices with privacy and usage cost concerns (regulations and business models have also driven adoption [115]). Smartphones are also natively third-party programmable (by our definition), and the wide consumer awareness of "app stores" has made it almost an expectation that applications (or "apps") are available for installation in large numbers. While some smartphone platforms (such as iOS[1] and Maemo) rely on app store maintainers to serve as gatekeepers against malicious (or maliciously exploitable) apps, many others (such as Android, BlackBerry 10, and Windows Phone) have permission-based security to restrict the damage that such apps might cause. Sole reliance on gatekeepers has the drawback that "side-loading" of apps from another source is then more likely to be prevented by the platform vendor (as is the case with iOS[2]).

A number of different terms are being used for essentially the same concept of a permission. By our definition a *permission* is something that is uniquely named, and something that a program (or rather its threads of execution) may possess. Possession is required for a program to be allowed to take certain actions (typically to call certain system APIs), or perhaps even to be the target of certain actions (e.g., an Android app may not receive certain system messages without the appropriate permissions [61]). A common reaction to an attempt to invoke an unallowed operation is to trigger a runtime error, although the concrete mechanisms for reporting such errors vary between platforms.

By the term *permission-based security model* we simply mean a security model in which access control is heavily based on permissions. We assume at least API access control such that different permissions may be required for different operations; i.e., there is finer than "all or nothing" granularity in granting access to protected APIs. With judiciously chosen restrictions for sensitive APIs a permission-based security model can serve as a central platform integrity protection measure. Such a model can also help permission-savvy end users (even if they are in the minority [62]) avoid leakage of private data and malicious exploitation of functionality.

Users, operators, and regulators all get some genuine benefit from platform security measures. Software developers, however, tend to only be inconvenienced by them, unless their software specifically requires functionality that platform security happens to provide. There are restrictions in what features can be had in an app, and how apps can be deployed (during a test/debug cycle, or in the field). This can even motivate the maintenance of multiple variants of an app [84] depending on what permissions are grantable for which distribution channel.

For most platforms the permissions required by an application must be declared. Writing the declaration may not in itself be difficult, but permission requirements are sometimes poorly documented [61], and keeping permission information up to date is an extra maintenance burden. The burden can be significant particularly for applications [84] that both exercise many sensitive APIs, and also have variants with different feature sets.

[1] In iOS 6, there is a small set of privacy-related permissions with application-specific settings. Developers need not declare the required permissions.

[2] As of early 2013, end-user installation of iOS applications is only allowed from the official vendor-provided App Store.

We present an approach for inferring permission requirements for programs constructed out of a selection of components in a permission-annotated codebase. While it takes effort to annote all sensitive primitives with permission information, the up-front cost is amortized through reuse in new program compositions. We have implemented the approach as one use case for the research language Magnolia, designed to be statically analyzable to the extreme. Magnolia avoids dynamic features, but has extensive support for static "wiring" (or linking) of components. We argue that these characteristics combine to facilitate permission inference without undue restrictions on expressivity. Magnolia also supports cross-platform code reuse, as its interface and implementation specifications allow for declaration of permission information in such a way that different platform-specific concrete permissions can be handled in an abstract way.

Magnolia is source-to-source translated into C++, and hence can be used to target platforms that are programmable in C++, including most smartphone platforms.[3] Translation to a widely deployable language is an important part of the overall portability picture, and also a possibility to abstract over differences in implementations of said language. Cross-platform libraries and Magnolia's support for interface-based abstraction help with the API aspect of portability. A third aspect is support for integration with platform vendor provided tools, which remains as future work in the case of Magnolia.

Maintaining permission information together with source code should result in better awareness of possible runtime permission failures when programming, and also allow for various automated analyses of the permission requirements of programs and program fragments. Such analyses, particularly when used in ways that affect the construction of software (e.g., due to analysis-based generation of permission declarations, or even code modifications), could also aid in the discovery of errors in app permission declarations or platform documentation.

While our focus is on permissions, some of the techniques presented apply not only to *right* of access, but more generally *ability* of access. E.g., from the point of view of error handling it matters little if a runtime failure is caused by lack of camera hardware, or lack of permission to access it. There are platform differences in whether requesting a permission will guarantee its runtime possession, and also in whether it is possible to similarly declare a (software or hardware) feature requirement so that availability of the feature will be guaranteed after successful installation. For instance, specifying `ID_REQ_REARCAMERA` in the manifest of a Windows Phone 8 app will prevent installation on devices without a back-facing camera [126]. Given the similarities between permissions and feature requirements we sometimes use the term *access capability* to imply access ability in a broader sense than that determined by permissions.

4.1.1 Contributions

The contributions of this paper are:

[3] Magnolia is not a *symbiotic language* (i.e., a language designed to coexist with another one), however, and there is nothing in Magnolia that would prevent its compilation into other languages. Still, the current implementation only targets C++.

- We give a brief overview of permission-based security models of a number of current smartphone OSes, and survey the associated tooling (if any) for inferring required permission information for applications.
- We present a language-based solution for declaring permissions for APIs and inferring permission requirements for programs. The solution allows for cross-platform programming by exploiting the host language's support for interface-specification-level abstraction over different implementations.
- We discuss static analysis friendly language design choices that favorably affect permission inference accuracy, and argue that some of the expressiveness cost of the resulting lack of "dynamism" can be overcome by flexible static composition.

To evaluate the presented solution we have implemented it based on the Magnolia language, and made use of it in a small cross-platform porting friendly application that requires access to some sensitive APIs. We have organized the app codebase to facilitate growing it to target multiple different platforms and feature sets, probably with different permission sets for different configurations.

4.2 Permission-Based Security Models in Smartphone Operating Systems

Below we list distinctive aspects of the permission-based security models of a number of current smartphone OSes (more wholesome surveys of the permission and security models of some of the same platforms exist [4; 115]). We also discuss any permission inference or checking tools in the associated vendor-provided developer offerings. We provide a side-by-side summary of permission-related details of the platforms in Table 4.1. Due to the newness of Tizen (no devices have been released as of early 2013) and the similarity of its and bada's native programming offerings, we opt to exclude Tizen (but not bada) from the table.

Android allows for the definition of custom "permissions". Permissions have an associated "protection level", with permissions of the "dangerous" level possibly requiring explicit user confirmation; hence a developer defining such a permission should also provide a description for it, localized to different languages [3]. A "signature" level permission does not have that requirement as it is automatically granted to apps signed with the same certificate as the app that declared the permission. No tools for inferring permissions for an app are included in the Android "SDK Tools" [3] as of revision 21.1. There are two third-party permission checkers capable of statically analyzing the permission requirements of Android apps. The tools are named Stowaway [61] and Permission Check Tool [178], and they both report on over/underprivilege wrt manifest-declared permissions. Their accuracy is discussed in Section 4.7.

bada 2.0 The Eclipse-based IDE of the bada SDK 2.0.0 [154] incorporates an "API and Privilege Checker" [153] tool that checks the project for privilege violations (an API requiring a privilege group is used, but the

privilege group is not declared in the manifest) and unused privileges automatically during packaging, and optionally during builds. The tool is for *checking* privileges, and does not generate privilege group declarations for the manifest.

BlackBerry 10 (BB10) is notable in that (upon first running an app) a user may grant only a subset of the "permissions" requested in the corresponding "application descriptor file" [26], and it is then up to the app to react sensibly to any runtime failures caused by unpermitted operations. BB10 also has limited support for running (repackaged) Android applications, with a number of Android features and permissions being unsupported [25].

MeeGo 1.2 Harmattan access control makes use of traditional "credentials" including predefined Linux "capabilities", Unix UID and GID and supplementary groups, and file system permissions. Harmattan adds to these by introducing fine-grained permissions known as resource "tokens", as supported by the Mobile Simplified Security Framework (MSSF) [130]. Granting of credentials is policy-based, and consequently (as of early 2013 and Harmattan version PR1.3) app credential information is not shown to the user, either in the app Store or under installed Applications. The `aegis-manifest` tool performs static analysis of binaries and QML source. It generates a manifest file listing required credentials for a program, but may fail in exceptional cases. Dynamically determined loading (e.g., via `dlopen`) or invocation (e.g., via D-Bus) of code are possible causes for the static scanner failing to detect the full set of required credentials.

Symbian v9+ Symbian OS has had a "capability-based security model" since version 9 [93]. It is unusual in that both executables and DLLs have "capabilities". A process takes on the capabilities of its executable. Installation requires code signing with a certificate authorizing all the capabilities listed in any installed binaries; a self-signed certificate is sufficient for a restricted set of capabilities. Any loaded DLLs must have *at least* the capabilities of the process. There is a "Capability Scanner" plug-in for the Eclipse-based Carbide.c++ IDE that ships with some native Symbian SDKs; the plug-in is available starting with the Carbide.c++ release 1.3 [131]. The scanning tool presents warnings about function calls in a project's codebase for which capabilities are not listed in the project definition file. The tool is only able to *estimate* the required capabilities.

Tizen 2.0 The Tizen 2.0 SDK [167] release introduced a C++ based native application framework, which appears to have bada-derived APIs. The permissions in Tizen are called "privileges"; the set of permissions (and their naming) in Tizen differs from those of bada. Privileges are specified in a manifest file in an installation package, and there is no tool support for automatically inferring and generating the privilege requests. However, as with bada, the Tizen SDK includes an "API and Privilege Checker" [166] tool for checking for potential inconsistencies between specified privileges and APIs being used in an application. The tool may be enabled for automatic checks during builds or code editing, and it may detect either under or overprivilege.

	Android	*bada*	*BlackBerry 10*	*MeeGo 1.2 Harmattan*	*Symbian v9+*	*Windows Phone 8*
permissions:	open-ended set of "permissions"	predefined set of "privilege groups"	open-ended set of "permissions"	open-ended set of resource "tokens"	predefined set of "capabilities"	predefined set of "capabilities"
permission categories:	"normal", "dangerous", "signature", and "signatureOrSystem"	"Normal", "System"	N/A	N/A	"User", "System", "Restricted", "Device Manufacturer"	"Least Privilege Chamber"
auth:	depending on permission: automatic, user approved (all or nothing), signed by authority, or preinstalled	by vendor at time of publishing, based on *developer* "privilege level"	user approval; user may only grant a subset of requested permissions	by installer, depending on software source and policies declared in software packages	user approved (all or nothing), developer signed, identity-verified developer signed, or vendor approved	user approval (all or nothing)
assignment request:	recorded in a manifest in installation package	recorded in a manifest in installation package	recorded in a manifest in installation package	recorded in a manifest in installation package	specified in project definition; recorded in binary	recorded in a manifest in installation package
inference by tools:	Stowaway (3rd party), Permission Check Tool (3rd party)	"API and Privilege Checker"	none	aegis-manifest	"Capability Scanner"	none (for WP8 – "Store Test Kit" for 7.1)

Table 4.1: Smartphone platform permissions and tools support.

Windows Phone 8 (WP8) has a security model in which the kernel is in the "Trusted Computing Base" "chamber", and where OS components, drivers, and apps are all in the "Least Privilege Chamber" (LPC) [95]. Software in the latter chamber may only directly invoke relatively low-privilege operations, and only when in possession of the appropriate "capabilities". All capabilities are user grantable, and the requested capability set of each app is disclosed in Windows Phone Store; some capability requirements are displayed more prominently than others. "Hardware requirements" may also be specified, and an app is not offered for phone models not meeting the requirements. The Windows Phone SDK 8.0 does not contain capability detection tools for apps targeting WP8[4], nor (as of early 2013) are such apps programmatically capability analyzed during Store submission [126].

[4]Windows Phone SDK 8.0 has a Visual Studio IDE integrated "Store Test Kit" that may be used to inspect a Windows Phone OS 7.1 targeting app and list the capabilities required by it. Windows Phone OS 7.1 is not natively programmable by third parties, and hence not a smartphone OS per our definition.

4.3 The Magnolia Programming Language

Magnolia [24] is a research language that aims to innovate in the area of reusability of software components. Safe composition of reusable components requires strict specification of component interfaces—sometimes referred to as APIs (application programmer interfaces) if semantic content is implied. A description of an API in Magnolia is given using the **concept** construct; a **concept** declaration can be thought of as an incomplete requirements specification. It specifies one or more abstract **type**s, some operations on those types, and the behavior of those operations (in the form of axioms). Each **concept** may have multiple **implementation**s that provide data structures and algorithms that satisfy its behavior. Each **implementation**, in turn, may satisfy multiple **concept**s.

One kind of operation that may be defined in Magnolia is a **procedure**. A **procedure** has no return values, but may modify its arguments according to specified *parameter modes* [13]. Legal parameter modes include **obs** (observe; the argument is read-only), **upd** (update; the argument may be changed) and **out** (output; the argument is write-only) [14]. A simple **procedure** that only outputs to a single parameter may equivalently be defined in a more "sugary" form as a **function**, and regardless of choice of declaration style, invocations to such operations may appear in expressions.[5] The keyword **call** is used to invoke an operation as a statement. A **predicate** is a special kind of function yielding truth values, and taking zero or more appropriately typed expressions as arguments. A **predicate** application as well as **TRUE** and **FALSE** are *predicate expressions*, and more complex predicate expressions are built using logical connectives.

The notion of *partiality* of an operation, meaning that the operation is not valid for all values that its parameter types could take, is central to Magnolia. Such a restriction can be specified for an operation. In the API it takes the form of a **guard** [9] with a predicate expression, which may include invocations to **function**s and **predicate**s. The more fine-grained notion of *alerts* [16] is the corresponding partiality notion in implementations. Alerts are an abstraction over **pre**/**post**conditions and error reporting, and each partial function is tagged with a list of **alert** names and the corresponding conditions that trigger the alerts. The set of defined alert names is user extensible and partially ordered, possible to organize as a directed acyclic graph.

```
alert CameraAccessAlert;
alert NoCamera <: CameraAccessAlert;
alert NoAccessToCamera <: CameraAccessAlert;

predicate deviceHasCamera() = Permission;
procedure takePicture(upd w : World, out p : Picture)
  alert NoCamera unless pre deviceHasCamera()
  alert NoAccessToCamera if throwing PermissionDenied
  alert NoAccessToCamera if throwing CameraInUse;
```

Here the alert names `NoCamera` and `NoAccessToCamera` are specialisations of the alert name `CameraAccessAlert`. The procedure `takePicture` has three possible error behaviors. The precondition test calling the predicate `deviceHasCamera`

[5] In Magnolia, an expression always yields a single value; i.e., there are no multi-valued expressions such as `(values 1 2)` in Racket.

checks whether the device has a camera; if not, it would not be meaningful to use the procedure. The two other conditions have the same alert name, and are triggered by the procedure implementation throwing one of two exceptions.[6]

A **`program`** is a special implementation in that its operations are made available as "entry points" to a piece of software that is composed in Magnolia. The Magnolia compiler translates Magnolia code into C++ source code, and produces a command-line interface wrapper for the **`program`** through which the exported operations may be invoked.

Due to Magnolia's explicit static linking of components (as declared in source code), all data structures and algorithms corresponding to a **`program`**'s types and operations (respectively) are known at compile time. Programs are statically typed, and there is no subtyping or dynamic dispatch (as e.g. in the case of C++ **`virtual`** functions). There are also no first-class functions (or even function pointers) to pass by value for parameterizing operations at runtime; any such parameterization must be done statically by specifying concrete operations used to implement a concept.

The static nature of Magnolia means that the actual target of an operation invocation appearing in program code is always statically known. Due to this it is possible to tell whether calls to a given operation appear in a given program composition, and any definitions for operations that have no invocations may be dropped for purposes of optimization or full-program analysis. Still, even in Magnolia's case it is generally not possible to tell if an operation appearing in a program actually gets invoked, as relevant facts about program runtime state or how far execution gets to proceed are generally not known at compile time.

4.4 Language Support for Permissions

Here we design a way to model permissions (and more generally, access capabilities) in Magnolia. As we prefer to keep Magnolia's core language simple, again for ease of analysis, we want to avoid feature-specific language extensions where possible. In this case we can do so by mapping permissions onto the Magnolia alerts system. The syntax may not always be as convenient as it could be, but that could be fixed through superficial syntactic transformations; we do not consider alternative syntaxes here.

The execution of a program consists of operations on the program state, and we want to be able to determine the permission requirements of all operations appearing in Magnolia code. To allow for this the permissions must either be declared, or it must be possible to infer them based on the implementation of the operation (i.e., its body). Magnolia currently allows an operation to be implemented either in Magnolia or in C++; for the former we can infer permissions by examining the language, but not for the latter. Any permission requirements for C++ operations will therefore have to be declared.

Permission-protected operations are associated with requirements, i.e., preconditions, as dictated by the platform APIs. We can state the preconditions as **`alert`**s with predicate expressions, noting that a permission restriction gives

[6] In real-world code we might want different alert names to distinguish between errors of a transient (`CameraInUse`) and permanent (`PermissionDenied`) nature. On most platforms application permissions are fixed at install time.

us two separate concerns: (1) we want to know of the permission requirement so that we can request the permission, and hence try to prevent runtime errors; and (2) we want to be able to handle any related errors. For case (1) we want platform-specific permission names, while for case (2) we would like abstract, platform-agnostic *error* names, probably relating to the operation. The example in Section 4.3 had the latter kind of names, namely `NoCamera` and `NoAccessToCamera`.

For storing platform-specific permissions we essentially just want to have the predicate expressions as named properties of operations. Had we support for convenient scripting of compiler-assisted queries we would not necessarily require fixed, predefined naming, but might rather choose any descriptive name to use as a search key to find the relevant expressions. The built-in support for permission inference in Magnolia currently uses the name `RequiresPermission` for this purpose (as suggested in Section 4.6, it might sometimes be desirable to use other names). We use `RequiresPermission` to "tag" permission preconditions, and each permission appearing in a precondition is defined as a "dummy" **`predicate`**.

As such predicates merely represent static properties, they are not intended to actually trigger an **`alert`** at runtime. This can be ensured by treating `RequiresPermission` as special and not inserting a precondition check for it. A more general alternative is to define the predicates as **`TRUE`**, leaving any generated check as dead code. On most platforms we can assume that the program is only started if the declared permissions have been granted, but there may be reasons for not requesting all inferred-as-required permissions. Permission-related precondition violations are thus possible, and we want them trapped as declared for their platform-agnostic **`alert`**s. It may be more efficient to capture any platform-specific runtime "permission denied" error than to actually implement a sensible predicate that checks for possession of the associated permission.

The Magnolia compiler supports scavenging a **`program`** for its operations (which, as mentioned in Section 4.3, are known in Magnolia) and respective permission requirements, provided the operations' permissions are specified as suggested above. (This approach also generalizes to other access capabilities, e.g. Windows Phone hardware requirements.) The result is conservative, but can only err on the side of too many permissions, assuming correct annotations. One source of inaccuracy is the currently indiscriminate inspection of all operations. Any dead code elimination done by the compiler happens later in the pipeline; such optimization would be beneficial, particularly if data-flow sensitive.

The second source of inaccuracy comes from the way we build the result. Perhaps the most accurate way to represent the result would have been as a single predicate expression such as `BLUETOOTH() && CAMERA() && (ACCESS_COARSE_LOCATION() || ACCESS_FINE_LOCATION())`, built as a collation of the relevant predicate expressions. Currently, however, we just build a set of permissions such as {`BLUETOOTH`, `CAMERA`, `ACCESS_COARSE_LOCATION`}. This may produce suboptimal results, as concrete choices must be made between logical alternatives. Our current implementation produces a set, and picks the left choice from OR-ed permissions.

Platform-provided sensitive operations typically require a fixed set of permissions, but there are many exceptions that motivate allowing the use

of logical expressions to at least *specify* permission requirements, even if we do not always make optimal use of the specification. Let us consider the `LocationManager` class of Android OS. Its `getLastKnownLocation(String)` method requires either `ACCESS_FINE_LOCATION`, or at least `ACCESS_COARSE_LOCATION`, depending on the "location provider" specified as the sole argument. The `NETWORK_PROVIDER` supports both coarse and fine grained positioning, and no `SecurityException` should get thrown as long as either permission has been requested (and granted). If we implement a network positioning specialized version of the operation—perhaps named `getLastKnownNetworkLocation`—then we may declare:

```
procedure getLastKnownNetworkLocation(upd w : World, out l : Loc)
  alert RequiresPermission unless pre ACCESS_COARSE_LOCATION() ||
      ACCESS_FINE_LOCATION()
  alert LocationAccessNotPermitted if throwing SecurityException
  alert IllegalArgument if throwing IllegalArgumentException
  alert NotFound if post value == null;
```

We are using a platform-agnostic `LocationAccessNotPermitted` alert to allow permission failures to be handled portably. The Android-specific permissions we are stating as a predicate expression tagged with `RequiresPermission`. Other possible errors for the operation are also mapped to alerts to allow handling.

For other platforms we would probably require a different (native) implementation of the operation, also with different error-to-alert mappings declared similarly to the above. E.g., on Windows Phone a `UnauthorizedAccessException` typically gets thrown on permission errors, whereas on Symbian one can generally expect a Symbian-native *leave* (a form of non-local return) with the error code `KErrPermissionDenied`. Interestingly, there are APIs (such as those of the Qt cross-platform application framework) that have been ported to different platforms, but which still necessarily have platform-specific permission requirements. With such APIs one could have a single (native) implementation but multiple Magnolia declarations (with different **`alert`** clauses).

4.5 Experience with Application Integration

For trying out the solution we created a small software project named *Anyxporter* (Any Exporter) [23], with the goal of building a codebase that would serve as a basis for creating various programs for exporting PIM (personal information manager) data in different (probably textual) formats. We chose the PIM exporting theme for exercising permissions as: (1) there are a number of different data sources, possibly requiring different permissions; (2) different storage/transmission options for exported data would likely require further permissions; and (3) the idea of building a "suite" of programs should allow us to keep the permission requirements of each individual program reasonably small, which may make a user feel safer in installing a given variant (since the program does not ask for permissions to do anything other than what the user wants done).

Anyxporter currently includes only one proper PIM *data source*, for reading contact data. Its implementation requires the Qt Mobility Contacts API [132]. Said API is implemented [132] at least for Symbian (S60 3rd Edition FP1 and later), Maemo 5, and Harmattan, and also for Qt Simulator for testing purposes

(without real contact data). For targets for which the API is not available, we have also implemented a "mock" data source that yields fixed contact data, and this data source has proved useful in testing other components of the software.

Of the targets supported by Qt Mobility Contacts, Symbian and Harmattan have permission-based security models, and our discussion here focuses on them. On Harmattan using the Qt API to *read* contact data requires the `TrackerReadAccess`, `TrackerWriteAccess`, and `GRP::metadata-users` credentials, whereas on Symbian only `ReadUserData` is required; clearly, the Symbian implementation of the API is better in respecting the principle of least privilege.

The default output option is to save to a file, which for a suitably chosen filesystem location requires no manifest-declared permissions either on Symbian or Harmattan. Anyxporter also has initial support for HTTP POST uploads of output files, implemented in terms of Qt 4.8 networking. Qt 4.8 is mostly unavailable on our example platforms, but Internet access generally requires no credentials on Harmattan, and the `NetworkServices` capability on Symbian.

Formatting of data for output is done using Lua scripts, and we currently include an XML formatting option for contact data. A Lua virtual machine (VM) instance is used as the intermediate representation (IR) between the different input and output options; in principle, data of the same kind (e.g. contact data) could have the exact same Lua object representation, regardless of concrete data sources and output formatters. Through careful choice of enabled Lua libraries we are preventing Lua code from doing anything other than "pure processing"; it cannot access platform APIs or the file system, and hence should require no permissions (or analysis for inferring permissions) on any platform.

The various library components of the app, such as file system interface, contact data source and Lua script interface, are specified by **`concept`**s. The main app code is programmed against these concepts, so that it is independent of the target platform. The app code is unaware of the exact nature of the permissions, though it may make use of and handle generic *permission denied* alerts.

Each library component has multiple implementations, one for each supported platform, with each implementation specifying platform-specific permissions. For example, the plain streams-based file system interface uses the following permission predicates:

```
predicate CXX_FILE_CREATE() = Permission;
predicate CXX_FILE_WRITE() = Permission;
predicate CXX_FILE_READ() = Permission;
predicate CXX_FILE_DELETE() = Permission;
```

A particular version of the app is built by composing the main app code with the platform-specific library implementations:

```
program CxxEngine = {
  use Engine; // application logic
  use CxxFileSys; // generic C++ versions of the library components
  use CxxLuaState;
  // use the 'mock' data source
  // the data source mapper will apply 'exportEntry' to each data entry
```

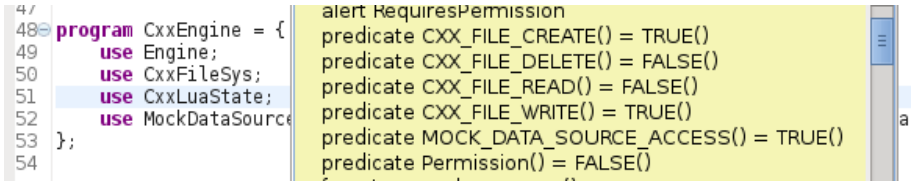


Figure 4.1: Hover information for a program in the IDE shows which permissions are enabled and disabled.

```
  use MockDataSourceMapper[map => mapDataSource, Data1 => File, Data2
      => LuaState, f => exportEntry];
};
```

Our system collects all the permissions used by `CxxEngine`, defines the value of the relevant predicates to be **`TRUE`** (and the predicates for the unused permissions to be **`FALSE`**), and then outputs the permission list in a text file, together with the C++ code for the program. Figure 4.1 shows an IDE display with the inferred permission requirements.

4.6 Problematic Permission Requirements

It is a Magnolia philosophy that incomplete specifications are okay, and that specifying as much as is convenient is likely to give a good return for effort. Documented platform permission requirements are generally straightforward for individual operations, and it is unfortunate if they do not directly translate into code, as one must then expend effort to considering how to best specify them without harmful inaccuracies. There are real-world permission requirements whose accurate and convenient specification challenges our design.

It is not uncommon for the permission requirements of a platform operation to depend on its arguments. Such requirements *can* be specified as a predicate expression for an **`alert`**, as shown by the example below. However, as argument values are generally not statically known, the operation is no longer guarded by a static predicate expression. Any permission analysis trying to determine the permission requirements of a program will then require a policy regarding how to translate such expressions to static ones without underprivilege or too much overprivilege. Perhaps a better alternative is to (where possible) divide the operation into multiple ones with static predicate expressions. We did so in a similar example in Section 4.4 by defining a location provider specific operation for a provider known to support coarse-grained positioning.

```
procedure getLastKnownLocation(upd w : World, out l : Loc,
                               obs p : Provider)
  alert RequiresPermission unless pre ACCESS_FINE_LOCATION() ||
   (supportsCoarse(p) && ACCESS_COARSE_LOCATION());
```

We have discussed declaring different permissions for different platforms, but there are also permission differences between different releases of the same platform. On Android, the permissions for some operations have changed

over time due to subtle and innocuous code changes [5] in their implementation. As such changes tend to only affect relatively few APIs and operations, it may be inconvenient to have to give separate **implementation** declarations in these cases. One possible, pragmatic solution may be to give different **alert** clauses for different platform releases. For example, we might generally specify AndroidPerm alerts for Android, but in some [5] cases use release specific alerts:

```
alert AndroidPerm8 <: AndroidPerm; // Android 2.2 (API level 8)
alert AndroidPerm9 <: AndroidPerm; // Android 2.3 (API level 9)
procedure startBluetoothDiscovery(upd w : World)
  alert AndroidPerm8 unless pre BLUETOOTH()
  alert AndroidPerm9 unless pre BLUETOOTH() && BLUETOOTH_ADMIN();
```

4.7 Related Work

Most of the literature on permissions is focused on Android, while our approach is to exploit the abstraction facilities of Magnolia in order to create platform-agnostic solutions. In Section 4.2 we already mentioned Stowaway [61] and Permission Check Tool [178], tools for analyzing the permission requirements of Android apps statically. As both tools are geared towards checking already declared permissions against code, the issue of deriving a concrete set of permissions to declare is perhaps less prominent; as explained in Section 4.4, the Magnolia compiler requires a policy for resolving logical permission expressions into sets.

Both Stowaway and Permission Check Tool resort to heuristics due to complexities of language and execution environment; heuristics-demanding complexities relating to language should not arise in the context of Magnolia. Stowaway's analysis appears more comprehensive than that of Permission Check Tool in that it attempts to handle reflective calls and Android "Content Providers" and "Intents". Magnolia has no reflective calls, and we propose that permissions be declared for *all* external-facing interfaces. Permission Check Tool works by analyzing source code using Eclipse APIs, whereas Stowaway takes Dalvik executable (DEX) files as input; the Magnolia ideal is to have programmable language infrastructure for custom analyses of semantically rich source code.

The Stowaway authors tackled poor platform documentation by determining Android 2.2 API permission requirements through API fuzzing. The PScout [5] tool has been found to discover more complete Android OS permission information. It performs a static reachability analysis between Android API operations and permission checks to produce a set of required permissions for each operation. Like our permission inferrer, PScout does path-insensitive analysis on source code. PScout's policy for "expression-to-set translation" is to take the union of all appearing permissions, which is more conservative than ours.

PScout has been used to extract permission specifications for multiple versions of Android. We are not aware of such analyses for other OSes, and problems of poor API documentation are compounded for cross-platform programming. With a suitably accurate and complete permission map available for a platform, one might imagine annotating a primitive with its set of

sensitive operations rather than its permission requirements, allowing for the latter to be inferred.

The kind of variability imposed by access capabilities is commonly handled using feature models [19; 22]. As shown in Section 4.4, access capabilities are associated with specific operations of an API, thus letting us use the alerts system of Magnolia for modeling their variability.

nesC [77] is a prominent example of a programming language with a programming model that is similarly restricted as that of Magnolia. Like Magnolia, nesC does static wiring of components so that types and operations become known at compile time; nesC even performs static component *instantiation* to avoid the overhead of dynamic memory management. The static nature of the language gives rise to a number of possibilities for accurate program analysis. E.g., the nesC compiler itself performs static whole-program analysis to detect data races. As nesC code is amenable to such analyses and the language also features interface-based abstraction support, we believe it would be a suitable substrate for a cross-platform permission inference solution. However, permissions are not applicable to TinyOS programming, which presently is nesC's primary domain.

As demonstrated by tools such as VCC [37], even unsafe languages (such as C) can be made static analysis (or verification) friendly with a suitably structured programming style and the addition of semantic information in the form of annotations. Additional annotations could also be used for permissions. Annotating an existing language is a valid implementation strategy for an analyzable language, with the advantage of avoiding another, full language layer. Magnolia's ground-up design for analyzability is likely cleaner, and the language can also be used merely as a tool for assembling programs out of C++ components.

4.8 Conclusion

Permissions are among the nuisances that software developers have to deal with. Language-based technology cannot lift access control restrictions, but it can help manage them, and reduce the chance of uncleanly handled permission errors occurring. Appropriate tools support enables automated analyses for determining a set of permissions that (if granted) will mean that no permission-caused runtime failures will occur. Suitable language can also help handle runtime failures in a portable manner, using abstract, concept or operation specific (not platform specific) permission failure reports and handlers.

We have presented such language and tools support. Our design relies on the base language taking care of: enforcing a programming style that does not prevent accurate static reachability analysis; and encouraging interface-based abstraction. Mere ability to declare permission information in a language is not special, as many languages (e.g., Java and Python) even support annotations as a way to attach custom attributes to declarations.

In Magnolia, the base language of our implementation, we can use core language such as **`predicate`**s and **`alert`**s to express permission conditionality and errors. Cross-platform interfaces may be exposed as `concepts`, and different **`implementation`**s and/or **`alert`** declarations may be used to express platform differences. Coupled with tooling, code analyses (and also transformations)

can be performed based on such declared information and what it implies.

In Magnolia, "dynamism" can only be allowed in a controlled way for correct permission analysis, and even then only outside the language. Analyzable, "static" language can be sugar-coated with convenient syntax, but certain familiar constructs are not directly transferable to Magnolia; e.g. a "traditional" higher-order `map` operation cannot be defined as functions cannot be passed as (runtime) arguments. Magnolia therefore carries some cost to expressiveness and developer familiarity, but offsets that by offering rich compile-time semantic information. Different language design tradeoffs could probably be made, while still allowing for accurate cross-platform permission inference. We see value in exploring awareness creating and preventative measures against potential software failures, whether caused by access control restrictions or other reasons.

Acknowledgements

We thank the anonymous referees for insightful comments on a draft of this paper. This research has in part been supported by the Research Council of Norway through the project DMPL – Design of a Mouldable Programming Language.

CHAPTER 5

Error Handling

Systematic management of permissions, as discussed in the previous chapter, may help avoid permission-related software failures. There are other kinds of failures, however, and the *possibility* of failures is pervasive in real-world software; failure management is a general concern in software development, one that must be taken care of in order to achieve robustness.

In constructing a software product line, one might standardize on a particular convention for error reporting, to be followed by the public APIs of different software components. Once a suitable convention has been chosen, language support for it can be devised, as necessary; mere APIs may not be capable of abstracting over the repetitive mechanics of realizing cross-cutting functionality such as error handling.

In this chapter we describe a language-supported failure management convention that permits syntax resembling those of mainstream exception mechanisms. We assume consistent adherence to it, so that the language need not have separate sub-notation and semantics only for code that might fail; this is beneficial for code stability, as one need not change notations where potential failures are introduced.

I believe that failure-management support in the style of this chapter could be adopted for a programming language that is used to implement platform-agnostic components, or to compose a software product out of components obeying the chosen error reporting convention. With this goal in mind, I designed the error propagation scheme to be such that it can be realized by emitting almost universally portable code. The scheme furthermore does not entail disruptive control flow nor dynamic features that would complicate or prevent inferring product configuration details (such as required permissions) by inspecting program code.

I presented an extended abstract on this topic at NWPT 2015 in Reykjavik [Hasu and Haveraaen, 2015]. This chapter contains a more code and language focused variation of a paper to be presented at NIK 2016 in Bergen [Hasu and Haveraaen, 2016].

Declarative Propagation of Errors as Data Values

Tero Hasu Magne Haveraaen

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway

Abstract

A "thrown" exception is a non-local side effect that complicates static reasoning about code. Particularly in functional languages it is fairly common to instead propagate errors as ordinary values. The propagation is sometimes done in monadic style, and some languages include syntactic conveniences for writing expressions in that style. We discuss a guarded-algebra-inspired approach for integrating similar, implicit error propagation into a language with "normal" function application syntax. The approach accommodates language designs with all-referentially-transparent expressions, and syntactic conveniences resembling those of traditional exception handling mechanisms. Our language-based failure-management solution furthermore supports automatically checking data invariants and function pre- and post-conditions, recording a trace of any due-to-an-error unevaluatable or failed expressions, and in some cases retaining "bad" values for potential use in recovering from an error.

5.1 Introduction

Traditional error handling mechanisms include explicit checking and propagation of error return values, as well as **`try`**/**`catch`**-style language constructs for intercepting non-local-returning, exceptional control transfers triggered by errors. The return-value-based idiom has the drawback of requiring extensive "manual code generation." It also suffers from a common problem in error handling, which is that most error information is volatile, with errors having to be checked in their immediate reporting context to avoid them going unnoticed.

The exception-throwing mechanism avoids those problems by transferring both control and error information over any code not capable of handling errors, but the difficulty of understanding potential error conditions and the consequences of such disruptive error reporting is a cause of trouble for both programmers and tools.

An unchecked exception triggers a non-local transfer (or even termination) of control as a side effect. Observable side effects result in the loss of *referential*

transparency [Strachey, 2000], i.e., a property of expressions that in essence means that replacing a subexpression by its value preserves evaluation semantics. Losing that property hampers equational reasoning about programs, as equivalences between expressions become unclear. Furthermore, potentially throwing operations and **`try`**/**`catch`** harnesses can induce numerous control jump sources and targets, adding to static analysis complexity, with the interdependent nature of data and control flows as an aggravating factor [Bravenboer and Smaragdakis, 2009; Liang et al., 2013].

One might argue that avoiding exceptions does not make much of a difference: as current mainstream languages assume extensive use of mutable values, aliasing, and side-effecting operations, most static program analyses for those languages have to be quite conservative in any case (as getting exact results would be intractable or uncomputable). In these languages, then, there may be little to gain in trading away expressive power for static semantics. There are other languages, however, that make different tradeoffs in order to support static reasoning. The research language Magnolia [Bagge and Haveraaen, 2010], for example, features a referentially transparent expression language, and would benefit (in terms of consistent design and richness of features) from having an error handling mechanism to match.

Existing approaches from functional languages already show that it is possible to define sub-languages that are referentially transparent and implicitly error propagating (beyond their original reporting context). The `Either` type constructor in Haskell, for example, is sometimes used in defining types for function results which are *either* successfully computed, or contain information about a failure. As `Either` is also a `Monad`, one can use Haskell's **`do`** notation to write an expression that implicitly detects and preserves any failure result of one of its sub-computations. The syntactic convenience of using monads implicitly is limited to **`do`** blocks, however, and there is no single error monad being consistently used for error reporting in Haskell (the language also features more traditional exceptions as a competing mechanism).

In this paper we explore the idea of making automated, monadic-resembling error processing the default in a language, in a way that makes it possible to retain referential transparency of expressions. Our solution's language-integrated error processing is not error monadic, but resembles it in that wrapped values (with error information) and implicit actions between operations are involved.

We furthermore explore a declarative language for adapting to different error reporting conventions. Operations need not be (re)programmed to report errors as wrapped data values; rather, code for checking for errors and converting to wrapped values can be inserted automatically. Our declarative language is based on the previously presented *alerts* [Bagge, 2012; Bagge et al., 2006] error abstraction, designed to abstract over different concrete mechanisms; consequently, in our solution pre- or post-conditions, exceptions, etc. can all trigger alerts, and be treated uniformly from an error handling point of view. Guarded algebras serve as the formal basis for our declarative checking of failure conditions.

For our error-information-enriched data types we have experimented with somewhat unorthodox information content, such that an error value may contain ① a history of expressions that failed to compute and ② any bad values produced by failed expressions. Both kinds of information may be

useful in error recovery. The actions required for history recording are not expressible as an error monad (which cannot *accumulate* error information), but they can nonetheless remain implicit through other forms of abstraction, as we show in this paper.

We have tried the presented error recording and propagation scheme with variants of a custom, small, purpose-built programming language named Erda. Erda$_{GA}$ is a dynamically typed functional language targeting the Racket virtual machine (VM), while Erda$_{C++}$ is a statically typed "first-order functional" language that compiles to C++ source code. The implementation of Erda$_{GA}$ is purely based on program transformations expressed in terms of the host language's macros, whereas Erda$_{C++}$ additionally relies on a compiler for deployment via C++. Differently oriented, implemented, and deployed language variants give some confidence in our scheme's generality and portability.

5.1.1 Contributions

The main contributions of this paper are:

- We show that propagation of errors as ordinary data values (of generic wrapper types) can be made implicit by realizing it at the language level so that all the relevant language constructs check and process additional ranges of data values.[1]
- We show that conversions between bare and wrapped values can be automated (where necessary, and without distinct expression syntax) based on declarative specifications, and explain how those specifications relate to guarded algebras.
- We suggest that the wrapped values might include a history of failed expressions[2] (which failed to compute a "good" result), and show that such history can be useful for deciding on recovery actions. We also show that a language can support retrying of any failed expression based on such history, and that this can be useful in implementing recovery actions.
- We suggest that wrapped values might also contain any "bad" values computed by fully-yet-unsuccessfully evaluated expressions (i.e., values that fail to fulfill a predefined invariant), and illustrate that such values can also be useful for recovery.

5.2 Guarded Algebras

The theory of *guarded algebras* [Haveraaen and Wagner, 2000] is a systematic approach to handling partiality and error values in algebraic style specifications.

[1] This idea has been seen before in monadic APIs, also at the language level when it comes to Haskell's **do** blocks and similar notations; we consider it more pervasively, however, not limited to special constructs in a language. Unlike the **do** notation, our solution also permits *direct-style* [Danvy, 1994] programs, in which function applications nest normally (without binding of intermediate results or passing of continuation arguments).

[2] It is common for a language to record a stack trace, or for a program to record a "cause" chain of errors, but here we consider recording information about the failed *expressions* and their arguments.

The approach allows total models and partial models, e.g., non-termination, yet allows the simpler total algebra reasoning in both cases.

The basic idea of algebras is centered around signatures and models for the signature. A plain *signature* (interface declaration) is a set of atomic type declarations, including the type `predicate`, and a set of function and predicate declarations, where the arguments and results are from the set of types. A predicate is a function with the return type `predicate`. Every type has an equality predicate. A *model* or *algebra* for a signature provides:

- A set for each type, called the *carrier set*, where the type `predicate` has a carrier set containing at least {`TRUE`, `FALSE`}.
- A total set-theoretic function for each function declaration, where the argument and result sets are the corresponding carrier sets.

For every signature we get the notion of type correct expressions with typed formal variables.

A *guarded signature* is a signature such that it in addition provides:

- A predicate `good` for each type, separating the carrier's *good* (when the `good` predicate evaluates to `TRUE`) and *bad* values (otherwise). The carrier elements `TRUE` and `FALSE` for the `predicate` type are its good values.
- A function declaration may have an explicit guard, i.e., a predicate expression that identifies its good arguments (when it evaluates to `TRUE`), as a subset of the good values of its argument types.

There is a well-formedness constraint ensuring that a function cannot appear directly or indirectly in its own guard.

We get the plain version of a guarded signature by disregarding the guards for its function declarations. The expressions for a guarded signature are the same as those for its plain signature.

On a computer a type is defined by a data structure, and its carrier is the set of bit patterns admitted by the data structure. The good bit patterns are those we consider valid for a type, e.g., as given by a data invariant. For instance, for ISBN numbers or the Norwegian national identity number system (*fødselsnummer*) this means having valid checksums; for C and C++ unsigned integer types all bit combinations are valid; and for IEEE floating point numbers [IEEE, 2008] this means not being a `NaN` (*not a number*).[3]

The guard for a function can intuitively be thought of as its precondition. Normally we are only interested in how functions behave on the good values, and so every function gets an implicit guard ensuring that it only receives good values.

Some functions are *handler function*s, however, possessing the ability to recover from (some) errors: in addition to taking its good arguments to good values, a handler also takes some (or all) of the bad values of its argument data types to good values. For example, a handler might try to turn a bad value into a good one based on redundant information in the bad data, such as that of a fødselsnummer. The fødselsnummer is an 11 digit number designed to allow for error correction, i.e., it is possible to find the correct fødselsnummer from a number with a few errors of the kind humans often do when handling long digit sequences by hand (swapping digits, misrepresenting a digit, etc.).

[3] For instance, a `NaN` is returned for the floating point operation 0/0, but not for 0.1/0, which instead yields a positive infinity value.

In many cases the recovery action is dependent on the local context that causes the error. For instance, the two function definitions $r_1(x) = \sin(\pi x)/x$ and $r_2(x) = x/(\sin(\pi x) + x)$ both yield a `NaN` (0/0 error for $x = 0$), but in the former case we should recover to the value π, and in the latter to the value $1/(\pi + 1)$, as given by L'Hôpital's rule.

5.2.1 Algebraic Specifications

An *algebraic specification* consists of a signature, giving the vocabulary, and a set of axioms. An *axiom* is a predicate expression, which *fails* for a model if it is `FALSE` for at least one combination of elements in the carriers of the model; otherwise the axiom *holds*. A model *satisfies* a specification when all of the specification's axioms hold for the model.

A *guarded specification* is an algebraic specification with implicit *goodness* axioms. A *goodness axiom* ensures that the result of a function is a good value (by the type's `good` predicate) for its good arguments. Good arguments to a function are good values which are also accepted by its guard (and must be good values for every expression in the guarding predicate). Good values for an expression are values which maintain goodness through every subexpression of the expression.

A model *satisfies* a guarded specification when all of the axioms hold for all good arguments. Thus any bad argument is ignored by the specification. For instance, we can write a guarded axiom for division as `(a / b) * a = 1`, which automatically ignores the case when `b = 0` if the function `x/y` has `y ≠ 0` as a guard. In the plain version we would need to explicitly exempt that case by writing `b ≠ 0 ⇒ (a / b) * a = 1`.

The definition of satisfaction for guarded axioms ensures that they only restrict the behavior of a model on the good arguments. The behavior for bad arguments and bad values is not constrained by the axioms. The restriction to good arguments has a profound effect: axioms on guarded signatures can just ignore any problematic cases, which can become overwhelming to account for in plain specifications. Translating a guarded specification to a plain specification means ① removing the extra structure of the signature, ② making the implicit goodness axioms explicit (one per function, which must include checks for good arguments), and ③ changing each axiom to include checks for good arguments for all of its subexpressions.

Due to the nature of guarded algebras, writing out the implicit axioms does not add any information to the guarded specification. Actually, writing axioms with a goodness predicate does not add any constraint to a guarded algebra. Thus the goodness predicates can be kept hidden in a guarded signature, and only need to appear when we want to see the plain version of the signature and its specification.

If we use partial set-theoretic functions as models for function declarations, the above requirements on guarded algebras force the functions to be defined for all good arguments. A partial function may thus be undefined only for bad arguments.

5.2.2 The Structure of the *Bad* Values

We define two guarded algebras to be *care equivalent* [Haveraaen and Wagner, 2000] if they have the same sets of good carrier values and the same sets of good arguments for each function, and if the functions give the same results in both algebras for the good arguments. Thus care-equivalent algebras only differ in the selection of bad values, and the behavior of the functions on bad arguments.

Studying the class of care-equivalent algebras, we get the following properties:

- This class has initial algebras. For the total case this is the free term extension. For the partial case this is the algebra where the functions are undefined for all bad arguments.
- The class has no final algebra. Specifically this means that there is no canonical way of dealing with a single error element as *the* bad value, nor is there a canonical way of avoiding bad values.

An *initial algebra* has a unique mapping to any algebra. Obviously the partial algebra that only defines good values and arguments has a unique inclusion to every algebra possibly defining bad values. The free term extension algebra informally represents the complete *history of failed expressions* (or *error history*). Every function call with a bad argument is an element of the bad values. Then of course every function call with such a bad value as argument is a bad value in itself, and so forth. This algebra is canonical in representing all information about what happened from the moment a bad argument was used (i.e., a record of what would have been done had the error not occurred), whereas the all-good-arguments case yields the expected computation.

A *final algebra* has a unique mapping from every algebra. In the class of care-equivalent algebras, an algebra may have functions that can recover from bad values. An algebra that recovers from a bad value does not have a mapping to an algebra that recovers from a different bad value, nor to an algebra that does not recover from a bad value, such as the single error element algebra.

Handler functions computing good values from bad values have to obey the guarded specification on their good arguments, but are free to recover from bad arguments in any suitable way. In some cases they might replace the first bad arguments in the error history with good ones, and then replay the history on the replacement values.

5.2.3 Guarding Programs

An implementation for a plain signature defines an algebra for that signature, by means of the code defining the data structures (sets of bit patterns) for the types and the algorithms (computations on sets of bit patterns) for the function declarations.

When implementing a guarded signature, we need to provide defining code for the corresponding plain signature. Thus we need to provide a data structure and designate a `good` predicate for each type, and we need to provide code for each of the declared functions and predicates, including the `good` predicate for each type.

With this code in place, the implementation can be verified or tested for conformance with the guarded specification or its plain counterpart, both of which concern the implementation's behavior on good values. Such validation includes the implicit axioms for the preservation of goodness (i.e., good arguments yield good results). This ensures *reliability*: the functions behave as specified on good arguments.

The guards for a function are more difficult to interpret when implementing the guarded signature. Technically, the plain version of a guard includes checks for good values for each argument, checks for good arguments for all subexpressions of the guards, and the actual guard predicates themselves.

In languages which support precondition declarations (e.g., Ada-SPARK or Eiffel) the plain version of the guard can be provided as a precondition. Otherwise, tool support for preconditions may be developed (e.g., JML for Java). In both cases, the tool can insert precondition checks wherever needed. If no tool support is available, precondition checks can be inserted manually at the start of every implemented function. Precondition checking can be avoided if the condition can be proven to hold. For instance, if the implementation satisfies the implicit goodness axioms, it is easy to prove that only external inputs need to be checked for goodness[4], as all internally produced values will be good values.

Traditionally, a precondition violation causes the program to terminate early, or otherwise disrupts the normal control flow. Alternatively, we can encode each such violation as a bad data value, and keep the normal control flow of the program, for referential transparency. The bad data values are automatically avoided by the guards, ensuring *robustness* in the sense that corrupt data does not induce uncontrolled behavior, which in turn could lead to an application's security or safety being compromised.

An implementer may decide to use a weaker precondition than the full plain condition corresponding to a guard. For instance, the binary search algorithm requires a sorted data array to function properly. Still, a program dealing with only mostly sorted data arrays might first attempt a binary search in the data, and on failing to find the key that way, revert to a sequential search to make sure that the sought data has not been overlooked. In this scenario, the binary search function need not require a sorted array as a precondition; instead, its found item data type can admit an "item not found(key, data array)" bad value. Sequential search can then be done by a *handler* that passes a good value through, and recovers from an "item not found" by either finding the item and yielding the good item value, or by returning a good "no such item" value.

While we in some cases want to call a handler directly after a possibly failing function call, as with the "item not found" recovery, doing so later on in the program may also be useful. Particularly if a complete history of what went wrong is available as the free term extension (i.e., the initial bad values for the care-equivalent algebras), it should still be possible to recover in a non-generic way, accounting for the circumstances of the failure condition.

Many data types lack spare bit patterns to encode bad values; e.g., in C

[4] Checking of input data is very important, as corrupt data has been known to cause security problems; e.g., corrupt JPEG or TIFF images have caused arbitrary code execution in browsers and image processing software.

and C++ unsigned integers, all bit patterns correspond to valid numbers. A data type may also simply have too few spare patterns for encoding all the bad values we wish to differentiate; e.g., an extreme case of this is the free term extension of bad values. To cater for these situations we may need extensive changes to our data types, with corresponding upgrades for our functions.

5.3 Automatic Pervasive Error Handling

We have argued that we can achieve referentially transparent and pervasive error handling by injecting error conditions as bad values into a data type. In our approach, normal functions will ignore the bad data, and special handler functions can deal with it at appropriate places in the normal control flow.

5.3.1 Automatic Bad Value Extensions

We define an *error extension* $\hat{T}_E$ of a normal type T as a disjoint union data type[5] with *cases*: `cgood`, containing only the good values of type T; and `cbad`, containing the error values E. The type E is chosen according to what error information the programmer wants to record. Functions f defined on T will have to be extended to functions on $\hat{T}_E$, while obeying the same guarded axioms. With tool support, we will be able to almost transparently deal with error extensions from normal code on T.

Error history extensions. The simplest error history extension is the free term extension, which defines the type E to be all terms (abstract syntax trees) with bad arguments. Thus the carrier for E is all terms representing expressions with argument values (encapsulated as terms) that cause evaluation errors, and all subsequent function calls having any such value as an argument.

Every function f defined on the normal types T gets extended to a function $\hat{f}$ on the encapsulated types $\hat{T}_E$.

$$\hat{f}(\hat{a}_1, \ldots, \hat{a}_k) \quad = \quad \begin{cases} \texttt{cgood}(f(a_1, \ldots, a_k)) & \text{if } \hat{a}_1 = \texttt{cgood}(a_1), \ldots, \\ & \hat{a}_k = \texttt{cgood}(a_k) \text{ and } a_1, \ldots, a_k \\ & \text{are good arguments for } f, \\ \texttt{cbad}(\text{`}f(\hat{a}_1, \ldots, \hat{a}_k)) & \text{otherwise.} \end{cases}$$

The notation $\text{`}e$ creates a term from the expression $e$. Note that $\hat{a}_i$ can be: ① a `cbad` value holding a part of the error history, which then gets extended by the $\hat{f}$ function call; ② an encapsulated value belonging to the `cgood` case yet forming a bad argument to the function $f$, in which case the entire expression becomes a `cbad` value containing the call and its argument values; or ③ an encapsulated value belonging to the `cgood` case and forming a good argument, in which case $f$ is evaluated and the result becomes an encapsulated `cgood` value. The extended function $\hat{f}$ will obey the same guarded axioms as f.

An error history extension E can also record additional information alongside the term, like the good or bad value from evaluating the guard, or a possible data invariant breaking value returned when calling f. This extra information is in principle reproducible from the other error history (assuming

[5] These are called algebraic data types in the context of functional languages.

that no external state is involved), making the extended E isomorphic to the free term extension. However, such extensions may simplify the implementation of handler functions.

User-defined error extensions. An extension of this kind is specified like the error history extension above, except that the user provides the type E, and defines a function `error_encoding` that will compute values in E from the term $`f$ and an argument list $(\hat{a}_1, \ldots, \hat{a}_k)$.[6] This can be useful if the errors for a type can be classified into a few cases. E can then cover those cases, and nothing more, possibly saving significant resources over the full error history, yet providing some details about the error.

Singular error value extensions. This is the classical case where a type is extended with a single error value, the type E becomes the singleton set, and the `cbad` case only indicates an error but no more detail about its origin.

Handler functions. A handler $\bar{f}$ is a function defined directly on $\hat{T}_E$, and as such it can recover from bad arguments while behaving as a normal function on the good cases. Handlers extend a function f on T with extra code for bad arguments, where f is a normal function (e.g., the identity function for a recovery-only handler). Since the $\bar{f}$ extension handles the good arguments like the automatic extension $\hat{f}$, it will obey the guarded axioms for f. Handlers based on identity functions can be placed anywhere in a program's code without altering its reliability (as it will still obey the same guarded specification), but may radically improve the well-behavedness of the program on bad input data (which may get turned into *formerly* bad data).

How to best recover from errors may depend on the local context where the error appears; e.g., consider the functions r_1 and r_2 in section 5.2. One way of recovering contextually is to place a specifically chosen handler function immediately after the expression that may cause an evaluation error. More generic handlers can be placed further away from the possible error sources, allowing a uniform or more centralized handling of a larger set of errors.

5.3.2 Bad Values from Run-Time Errors

In order to do an error extension $\hat{T}_E$ for functions (or predicates) f on types T, we need to understand what error reporting mechanisms are used when calling f on bad arguments from T. Hardware architectures and software engineering practices have many idiomatic ways of alerting software about such error information. We follow Bagge et al. [2006] in suggesting how we can translate the various alerting idioms systematically into our idiom of instantiating a `cbad` value in E.

Preconditions. A *precondition* is a predicate defining the good arguments for a function. If the precondition predicate applied to the arguments does not yield `TRUE`, the arguments are bad and the term must be encapsulated as a `cbad` value.

[6] In effect, the function `error_encoding` is a "homomorphism" from the error history extension to the user defined type E.

Postconditions. A *postcondition* is a predicate providing a specification of a function's result. A postcondition violation (the predicate does not yield `TRUE` on the result of f), can be interpreted as identifying bad arguments to f, hence the call should be encapsulated as a `cbad` value, possibly also including the computed violating result value. Such an approach is convenient if explicitly checking for good arguments is (almost) running the algorithm of f, while the postcondition predicate is a simple test on f's result.

Data invariants. A *data invariant* is a predicate provided with a data structure, where the predicate discerns between good values and inconsistent values. All values in `cgood` T are assumed to yield `TRUE` when tested by T's data invariant. The invariant is therefore assumed to be both an implicit precondition and an implicit postcondition for functions f, and can be treated as such for the purpose of creating $\hat{T}_E$ values. However, often the implementation of data invariant preserving functions may be split into helper functions. Helper functions may create an inconsistent value from a consistent or inconsistent value in T, or consume inconsistent T values and yield a consistent value in T for actual use. A helper function g of the former kind will be encapsulated as a $\hat{g}$ function yielding (accumulated) $\hat{T}_E$ error values for postcondition violations. A helper function h of the latter kind should be turned manually into a handler function $\bar{h}$ that takes $\hat{T}_E$ error values to $\hat{T}_E$ `cgood` values.

Error codes. An error code is a good value outside of the normal return value range of a function, used by the function to signal an error. Error codes can be captured in the same way as postcondition violations.

Error flags. Error flags are extra output arguments recording the state (OK, error, and possibly what kind of error) of the result of a function call. They behave like error codes, and need to be checked before the function's return value is used. Error flags can be captured as postcondition violations: the flag has to OK the returned data, otherwise the function received bad arguments.

Status variables. Status variables are similar to error flags, except that they are global variables being set by the failing function. Normally a status variable is reset by the next function call, but sometimes the status variable is sticky and has to be explicitly reset. We can handle this by checking the status variable after each return, resetting it and treating its value similarly to the error flag. As an optimization, sticky error flags allow the granularity of checking to be reduced.

Exceptions. An exception mechanism causes a disruption of the normal control flow when an error occurs. In our setting, we need to catch any exception and translate it into its corresponding `cbad` value. The computation can then follow the normal path for either case of $\hat{T}_E$, improving code reasoning, simplifying code transformation, etc.

5.4 Erda

We have described error handling automation for bad-value extending types and translating between alerting idioms. That automation relies on programming language support, and we have implemented variations of such support for our proof-of-concept *Erda*[7] language family. For this paper's concrete code samples, we mostly use the Erda$_{GA}$ language, whose native error processing behavior matches the error history extension semantics of section 5.3.1.

5.4.1 Erda$_{GA}$

Erda$_{GA}$ (*Er*ror *da*ta *G*uarded *A*lgebras) is a dynamically typed language implemented purely in terms of Racket's "normal" language implementation facilities, by macro expansion to Racket's core language, which is then executable directly in the Racket VM. In Racket, a language is normally implemented as a library whose exported macros define the surface syntax of the language [Tobin-Hochstadt et al., 2011].

Erda$_{GA}$ reuses Racket's module system directly, and bears a close syntactic resemblance to Racket. Despite the similar looking syntax, Erda$_{GA}$ is different in that its expressions take and yield good-or-bad wrapped values by default; all literals, for example, get an implicit `Good` wrapper. The wrapper data type is effectively an algebraic data type, with the `Good` and `Bad` variants defined as Racket structure types, sharing a common, "abstract" `Result` supertype.

In Erda$_{GA}$, a failed-expression history is also built out of `Good` and `Bad` objects. More specifically, a history is an AST with `Bad` branch nodes and `Good` leaf nodes. A `Good` node represents an atomic value with no history, whereas a `Bad` node represents a function application. A `Bad` node has further AST substructure, namely the applied value and its argument list, where the applied value would usually be a function (as required for its successful application). This simple design of only two AST node types works since *all* `Bad` values are created by some operation, even if it is just an explicit error instantiation (e.g., with **`raise`**).

Not all Erda$_{GA}$ expressions are function applications. Expressions like literals and variable references never fail, and thus cannot yield a `Bad` value. Expressions like **`begin`** and **`let`** do not in themselves produce a value, and those expressions are not included in history; only their value-giving subexpressions are.[8] Erda$_{GA}$ expressions that may (directly) produce a bad value, and which are not function applications, are turned into function applications, at least for purposes of history recording. For example, all conditional constructs (e.g., **`if`**, **`and`**, **`or`**, and **`cond`**) are macros that translate to applications of Erda$_{GA}$'s ***`if-then`*** function, whose "then" and "else" arguments are wrapped in a **`lambda`** to delay their evaluation.

Erda$_{GA}$'s history representation scheme allows error-history-extended operations (as described in section 5.3.1) to be introduced into the language purely in terms of macros and functions, without requiring a mechanism to introduce new AST node types; defining a new operation mostly just means

[7] Documentation: `https://bldl.ii.uib.no/software/pltnp/erda.html`

[8] Discarding of *side-effecting* sub-expressions may mean that history replay does not reproduce all the original behavior of an expression.

ensuring that any associated function calls consistently obey the expected convention of processing good and bad values. For Erda$_{GA}$ functions, automated bad value extension is done at the definition sites, by inserting additional code into the function bodies.

A function may be defined with Erda$_{GA}$'s ***define*** syntax, which accepts an optional `#:alert` clause for specifying guards for the function. The breaking of a function application's guard automatically results in a `Bad`-wrapped value, which makes it rarely necessary to explicitly construct bad values in code. The `#:alert` syntax closely resembles that of the original alerts proposal [Bagge et al., 2006]:

```
(define (div x y) #:alert ([div-by-0 pre-when (= y 0)])
  ; call Racket's division operator /, assumed imported as rkt./
  (rkt./ x y))
```

A **define**d function by default has an implicit guard enforcing all-good arguments. The optional `#:handler` modifier overrides this behavior so that it becomes possible to define a function able to handle (some) bad arguments. For example, we can write a function that accepts *any* argument (even a `Bad` one), and returns `(Good 42)`:

```
(define (forty-two x) #:handler 42)
```

Erda$_{GA}$ features a transparent foreign function interface (FFI) mechanism, allowing Racket functions to be called directly, with automatic unwrapping of arguments and wrapping of results (we relied on that in implementing `div`, above). It is not necessary to declare total Racket functions, but any partial[9] ones should be ***declare***d, using `#:alert` clauses to specify how errors are reported (so that, e.g., an exception gets caught and converted to a `Bad` value):

```
(declare (/ x y) #:is rkt./
  #:alert ([div-by-0 on-throw
              exn:fail:contract:divide-by-zero?]))
```

It is the **declare** form that creates an error history *extension* $\hat{f}(\hat{a}_1, \ldots, \hat{a}_k)$ for a primitive function f, as specified in section 5.3.1, whereas native Erda$_{GA}$ functions are implemented to process wrapped values to begin with. It may be worthwhile to **declare** Racket primitives even when it is not strictly necessary, as the history extension is then prepared as code at compile time, rather than performed dynamically. Erda$_{GA}$ is also able to generate more straightforward code for direct applications of **declare**d and **define**d functions.

In addition to the alert-supporting defining forms, Erda$_{GA}$ also includes language for constructing, inspecting, and modifying error values, to support the definition of handler functions for recovery or error reporting. Each alert has an inspectable name, for instance, represented as a Racket symbol. A simple example of creating a bad value, inspecting its alert name, and doing a name comparison is this expression evaluating to `(Good #t)`:

[9] In a dynamic language like Erda$_{GA}$ or Racket, any function expecting arguments of specific types is actually partial. One could consider enforcing required argument types with `#:alert` pre-conditions, but where a type error can be considered a programming error, one may want to terminate the program instead of producing a bad value. For type assertions, one might opt to use Racket's built-in "contracts" mechanism, for instance.

```
(alert-name=? (bad-result-alert-name (raise 'not-found))
              'not-found)
```

Error "Catching"

Although errors are just values, and inspectable and manipulatable using general-purpose language, it is still possible to introduce error recovery syntax that resembles conventional **try**/**catch** constructs (e.g., as found in C++ and Java).

Erda$_{GA}$ has a ***try*** construct as an error-handling-specialized "case" expression. Where a **try** form's body expressions produce a `Bad` result, the original error's alert name is matched against those specified for `#:catch` clauses. One can have multiple `#:catch` clauses, list multiple alert names per clause, and it is also possible to include a "catch all" clause.

In this example we define a function `add-creator`, ostensibly for adding information about the device from which a document object `doc` originates. Internally, the function uses **try** to recover from a potential API access permission error due to the `serialNumber` call (conversely, the `modelName` function is known not to fail). As a form of recovery, a note about any failure is added to a `notifications` object, which is also returned:

```
(define (add-creator notif doc)
  (let* ([hw (HardwareInfo)]
         [doc (add-creator-device-model doc (modelName hw))])
    (try
     (cons notif (add-creator-device-id doc (serialNumber hw)))
     #:catch
     [(NoPermission)
      (cons (add-missing-permission-note notif
              'read_device_identifying_information)
            doc)])))
```

For the common case of recovering from any error, Erda$_{GA}$ also has a ***::>*** form, as a more concise alternative to using nested **try** forms with `#:catch`-all clauses.[10] The `::>` form takes one or more subexpressions, which are evaluated in order until one of them gives a good result, or until the last one is reached. For example, we can use `::>` to concisely specify a default value for the case where a database `lookup` function triggers some alert upon trying to look up a value for a field that does not exist for the specified `entry`:

```
(format "Entry saved for ~a"
  (::> (lookup entry 'name) "Doe, Jane"))
```

Recovering by Replaying History

Since Erda$_{GA}$'s error history contains the function and argument values of failed operations, it is possible to retry an operation, possibly with some of the arguments replaced with different values. Erda$_{GA}$'s ***redo*** and related functions implement such history "replay." When coupled with a comprehensive API for

[10] The `::>` form is named after the `<::` operator of Bagge et al. [2006].

examining and modifying history, this facility opens up various possibilities for performing recovery.

For a basic, nonsensical example of recovering with history replay, we can rephrase the above database lookup recovery to work from *outside* the `format` expression itself. The `format` call should only fail if its second argument produces a bad (or unformattable) result, and thus we might retry a failed `format`ting by replaying the recorded call with a different second argument:

```
(let ([msg (format "Entry saved for ~a" (lookup entry 'name))])
  (::> msg
       (let ([args (bad-result-args msg)])
         (redo-apply msg (args-list-set args 1 "Doe, Jane")))))
```

The outside-recovery idea can be extended to recursively rewriting history rooted at a bad value. One possible application might be to work around a weakness in a third-party library written in Erda$_{GA}$, while still avoiding forking the code in that library. Suppose, for example, that we are using some social network client library for Android, whose `create-account` function wants to upload all of the client device's contact data upon creating a new account:

```
(define (create-account data)
  (define lst (read-all-contacts))
  (create-account+upload data lst))
```

We may remember to correctly request the `READ_CONTACTS` permission for our application, but those running a suitable version of CyanogenMod[11] may be using Privacy Guard to deny that permission for our application, causing `read-all-contacts` to fail inside the third-party code. To recover from a failed `create-account` call, we might `rewrite` its failure history to remove any operations that failed with a `NoPermission` alert, instead substituting an empty but good result. With that change made, the rest of `create-account` may well succeed when **redo**ne, making it seem that the user's contact database has zero entries:

```
(define (rewrite v) #:handler
  (if (bad-result? v)
      (if (alert-name=?
           (bad-result-alert-name v)
           'NoPermission)
          null ; empty contact list
          ; recurse into arguments stored in v
          (bad-result-args-map rewrite v))
      v))

(define account-id
  (try
   (create-account account-data)
   #:catch
```

[11] `http://www.cyanogenmod.org/`

```
    [(NoPermission)
     (redo (rewrite value))]))
```

Our `NoPermission` recovery in this example already hints at the **`redo`** mechanism possibly being useful for implementing centralized handling of known *kinds* of errors, wherever they may occur.

To expand on that idea, let us consider a control loop that selects commands (with some randomly chosen arguments) based on user input, and tries to deliver them to a Sphero[12] robot over a Bluetooth connection. Our `main` loop will only exit if it fails to `recover` from a bad `reply`, as discussed below:

```
(define (main)
  (define choice (user-choice)) ; read user input
  (define reply ; robot response or a failure
    (cond
      [(= choice 0) (set-stabilization (random-stabilization))]
      [(= choice 1) (set-rgb-led (random-rgb))]
      [(= choice 2) (set-back-led (random-color))]
      [#:else (set-heading (random-heading))]))
  (do ; section 5.5.5 explains do
    (recover reply) ; returns a good reply as is
    (main))) ; repeat only if recover succeeded
```

Our commands can *all* fail for the same reason, i.e., a missing or broken communication channel to the robot. The obvious recovery for such a failure is to `reconnect`, and then retry issuing the command. In Erda$_{GA}$, we can realize the retry functionality without explicitly keeping track of *which* command needs retrying, as we get that bookkeeping from the language. This makes it convenient to implement our centralized error `recovery` function, one that only fails if it cannot arrange to ultimately **`redo`** its argument `r`, as would happen in the event of an alert that it is unprepared to handle:

```
(define (recover r) #:handler
  (try r
    #:catch
    [(ConnectionFailed)
     (do (reconnect)
         (recover (redo r)))]
    [(StabilizationOff)
     ; set-heading requires stabilization to be on
     (do (recover (set-stabilization #t))
         (recover (redo r)))]))
```

5.4.2 Erda$_{C++}$

Compared to Erda$_{GA}$, the *Erda*$_{C++}$ language has a substantially different design and implementation. Syntactically the two languages look mostly identical, however, with the notable exception of Erda$_{C++}$ requiring additional annotations to support compilation to C++ source code; Erda$_{C++}$ programs are

[12] `http://www.sphero.com/`

fully, statically type resolvable. Erda$_{C++}$ is not a functional language, as its functions are not values; as they are not values, they also cannot be `Bad`.

For an example of Erda$_{C++}$ code, suppose we have a C++ implementation of a `factorial` function. We might ***`declare`*** the function's pre-condition and type with

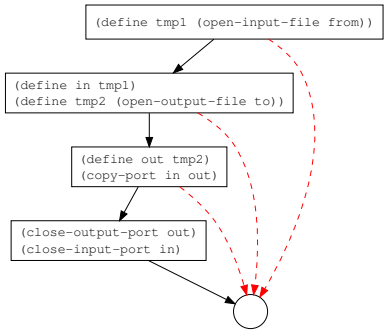
```
(declare (factorial x) #:: (foreign [type (-> int int)])
  #:alert ([negative-arg pre-when (< x 0)]))
```

Erda$_{C++}$ is also implemented as a Racket library, but the library's macros target a different core language, and the generated code's C++-based run-time environment is more limited than Racket's. More specifically, Erda$_{C++}$ targets the core syntax of *Magnolisp*, which is a language and infrastructure for complementing Racket's language definition machinery to support source-to-source translation into C++ [Hasu and Flatt, 2016]. Due to compatible core syntax, an unmodified Magnolisp compiler is capable of translating Erda$_{C++}$ code into C++. For example, below is a comment-annotated extract of C++ code generated for the Erda$_{C++}$ expression `(factorial x)`, where `x` is a variable, and `factorial` is a function declared as above:

```
if (is_Bad(x)) { // whether argument `x` is bad
  lifted = Bad(Nothing<Result<int>>(), AlertName("bad-arg"));
} else {
  // ....
  if (is_Bad(lifted_1)) { // whether (< x 0) failed to evaluate
    lifted_4 = Bad(Nothing<Result<int>>(),
                   AlertName("bad-precond"));
  } else {
    Result<int> lifted_5;
    if (Good_v(lifted_1)) { // whether (< x 0) holds
      lifted_5 = Bad(Nothing<Result<int>>(),
                     AlertName("negative-arg"));
    } else {
      int const r7 = factorial(Good_v(x)); // call the primitive
    }
  }
}
```

That C++ code reveals that Erda$_{C++}$ realizes the bad-value extension of primitive functions at call sites, which is another difference to Erda$_{GA}$. That difference allows for another one: Erda$_{GA}$ evaluates its arguments lazily, stopping argument evaluation at the first `Bad` argument, except for handlers. That difference further means that it is generally not possible to store all arguments for a failing operation. It is also not possible to store the function itself, as Magnolisp does not support function values or pointers. Erda$_{GA}$ does not implement failed expression history, but it still features alert-directed, implicit error triggering and propagation. Currently, only the alert name and any invariant-breaking value are incorporated into instantiated `Bad` values.

As Magnolisp only supports *abstract* user-defined data types, Erda$_{C++}$'s `Good` and `Bad` value constructors do not correspond to a record type (as in



(a) A potentially exception-throwing file-copying routine written in Racket. It is assumed that closing an opened port will not throw.

```
(define in (open-input-file from))
(define out (open-output-file to))
(copy-port in out)
(close-output-port out)
(close-input-port in)
```

(b) A file-copying routine written in Erda$_{GA}$, with only a single basic block. The code is sub-optimal in that the `out` port may get opened needlessly even if the `in` port was not. However, unlike figure 5.1a's routine, this version features safe resource cleanup.

Figure 5.1: The control flows of two routines of roughly the same functionality, with basic blocks shown.

Erda$_{GA}$). Rather, they are defined as operators constructing values of the abstract type (`<> Result T`), where `T` is a universal type parameter. In addition, for (`<> Result T`), Erda$_{C++}$ only requires some predicates and accessors for inspecting values of the data type. All of these operators are polymorphic in `T`, and implemented as primitives.

Erda$_{C++}$'s run-time library includes C++ implementations of the abstract data types required by the language itself (e.g., `Result<T>` and `AlertName`, which appear in the above listing). The library is made up of a single header file, with only the C and C++11 standard libraries as dependencies.

5.5 Discussion

Our approach provides an API's implementor with a way to express their domain knowledge about the error behavior of functions. Any knowledge or speculation about potentially useful recovery actions can also be included in the API, as a selection of handler functions with some acceptable bad arguments, to give the API client some predefined recovery options to choose from (or not) depending on context.

One important aspect of error recovery is to ensure that resources are not "leaked" even under abnormal conditions. In our approach, the lack of non-local returns and automatic "bookkeeping" of successfully initialized objects can facilitate straightforward and clear resource cleanup; no separate mechanisms such as RAII[13] [Stroustrup, 1997, section 14.4] or cleanup stacks [Turner, 2014] are required, and the ordering of cleanup actions stays apparent. As illustrated in figure 5.1a, safe resource cleanup can read as simply as failure-unsafe code, while also being easier to reason about.

[13] Resource Acquisition Is Initialization

5.5.1 Portability

One of the goals of our failure management solution is for it to also serve as an error handling convention that is highly portable.

The bad-value extension of data types in itself places few requirements on the run-time language; most essentially, a conditionally-branching construct must be available for error checking, and it must be possible to define a suitable wrapper data type (or types). The Magnolisp core language targeted by Erda$_{C++}$ is very limited, yet sufficiently powerful: it includes an **`if`** core form; it supports abstract data types implemented externally in C++; and its type system supports generics, allowing one wrapper type definition to be parameterized in order to satisfy static type checking.

Allowing an operation that produced a bad value to be redone merely based on information in the bad value is more demanding on the run-time language, but not arduously so. Our experience with Erda$_{GA}$ shows that retrying an operation only requires the ability to store a function as a value (along with any free variables), together with its argument values, and that is possible in many languages. As long as all the relevant expressions can be formulated in terms of function calls, it appears not to be necessary to be able to reify an execution context as a delimited continuation [Felleisen et al., 1988; Felleisen, 1988], for example, although Racket would be one of the few languages that supports them [Flatt et al., 2007].

5.5.2 Code Size and Optimization

Realizing transparent processing of bad-value-extended data involves glue code whose size can vary heavily depending on how "dynamic" the code is, and to where it is inserted (and how many times).

Erda$_{C++}$ is at the "bloated" end of the code footprint scale, in that its macros generate relatively primitive code, avoiding features such as first-class functions, record types, and syntax objects, which are unavailable. Erda$_{C++}$ also inlines alert checking code at each call site; while this should provide intraprocedural optimization opportunities for further compilation as C++, it can also severely bloat the code, particularly as pre- and post-conditions may also involve alerting calls.

Erda$_{GA}$, in turn, generates relatively compact code. It does this by inserting alert-checking code at definition sites, either right inside natively defined functions, or into separate wrapper functions for primitives, meaning that there is only one copy of the glue code per function. Calls to unknown primitives in Erda$_{GA}$ do not require even that, as it is only necessary to modify calls to happen via a *shared* higher-order function that performs the required work dynamically.

Yet another approach to code generation might involve a target core language with special-purpose constructs for processing `Result` values. That kind of approach could be expected to reduce core language footprints, with the more error-semantics-aware language also potentially providing additional optimization opportunities during further processing.

5.5.3 Incorporating "Bare" Language

Erda$_{GA}$ (like all Erdas) defaults to doing implicit error checking and propagation, but does feature a mechanism for locally switching to "bare" values without implicit processing. That mechanism can be useful for example with IEEE floating-point number operations, which themselves support abnormal values[14], propagation of such values across multiple operations, and recording of sticky error status flags in out-of-band state.

Erda$_{GA}$'s ***let-direct*** expression unwraps its named argument values (provided that they are all `Good`), evaluates its body expressions without implicit error processing, and then wraps the result solely according to its data invariant.[15]

Using **`let-direct`** we can, for example, perform multiple IEEE floating point operations, and then only afterwards check for a not-a-number result:

```
(define (f a b x)
  #:alert ([NaN post-when (nan? value)])
  (let-direct ([a a] [b b] [x x])
    (fl* (fl+ 1.0 (fl/ a b)) x)))
```

In a post-condition, the name ***`value`*** gets bound to the function's result. Thus, with the above definition of `f`, the expression `(f 0.0 0.0 10.0)` triggers a `NaN` alert, as **`value`** in that case is a `Bad +nan.0` due to $\frac{0.0}{0.0}$ being undefined.

5.5.4 Error History Buildup, Capping, and Control

For some control flows, naive history recording can result in values that have unacceptably large history. One likely scenario of this happening is where a potentially failing function is called non-tail recursively. For instance, suppose we have an Erda$_{GA}$ function `sum-1/n` such that it incorrectly computes $\sum_{i=1}^{n} \frac{1}{i}$, forgetting to include a check for the lower bound; thus, instead of computing the desired result, `sum-1/n` builds error history whose size is linear in n:

```
(define (sum-1/n n) #:alert ([not-gt0 pre-unless (> n 0)])
  (+ (/ 1 n) (sum-1/n (- n 1))))
(sum-1/n 5) ; =>
  ; (Bad bad-arg: #<procedure:+> 1/5
   ; (Bad bad-arg: #<procedure:+> 1/4
    ; (Bad bad-arg: #<procedure:+> 1/3
     ; (Bad bad-arg: #<procedure:+> 1/2
      ; (Bad bad-arg: #<procedure:+> 1
       ; (Bad not-gt0: #<procedure:sum-1/n> 0))))))
```

In practice, it is probably necessary to cap history buildup in some way. Any history capping policy should account for language-specific factors regarding what parts of history may acceptably be lost, to avoid causing inconsistencies or surprises in using language constructs that rely on the information.

[14] Abnormal values include "subnormal numbers", infinities, and "not-a-number" values.

[15] The **`let-direct`** construct is implemented by having a "syntax parameter" [Barzilay et al., 2011] determine (at macro-expansion time) which implementations of Erda$_{GA}$ operations to use (i.e., ones using wrapped values, or not).

Where the information is merely shown as an error trace to the developer, it seems reasonable to drop entries from the middle; this is comparable to common practices in displaying stack traces. It might also be appropriate to drop or compress duplicates of items that repeat (e.g., `Bad bad-arg`, above). Existing logging solutions demonstrate other potential "log throttling" policy options that might likewise fit the programming language context.

When error history is actually used by a program to affect its own behavior, it would seem natural to give the program explicit control over history recording, to ensure that all the necessary history is retained for long enough. Interestingly, the `Result` values (also good ones) themselves might provide a possible channel via which to communicate such preferences in a fine-grained manner, with dynamic extent, perhaps in the form of a "contagious" flag to indicate that history should be recorded.

Particularly for a language like Erda$_{C++}$, which has a non-garbage-collected target, we might also wish to allow explicit control over memory management for error history. Rather than a mere flag, we might instead propagate a handle for a memory "region" [Tofte and Talpin, 1994, 1997], to use for allocating those values' history, allowing their history to be freed all at once, perhaps upon exiting the responsible error handler's scope. Such memory management with "explicit regions" [Gay and Aiken, 1998] can be both faster and more memory frugal than heap allocation or garbage collection.

5.5.5 Bad Value Extensions vs. Monads

A *monad* [Wadler, 1995] is an implementation of a certain kind of collection of operations. Monads serve as a popular way in functional programming to add sequentialization, and to implement language with type-directed semantics. One popular application is to implement implicit checking for bad values.

An *error monad* is a data type that represents either a "good" value, or information about an error. Its monadic primitives apply monadic functions only on good values; once one of the functions returns a "bad" monadic value, the subsequent ones are skipped, and the bad value is the result of the combined computation. This has an effect similar to a singular error value extension, although the error information data type need not be a singleton set.

Monads capture a fairly ubiquitous programming pattern, and as a common abstraction they can promote reuse of code, syntax, and tool support. A number of languages support "sugary" notation similar to Haskell's **`do`** syntax for writing monadic expressions, for instance, and there are also language extensions to assist in producing monadic code [Kotelnikov, 2014; Swamy et al., 2011].

Monadic sequencing is not a natural fit for implementing error history accumulation in our scenario, however. In our case functions should appear total, enabling programming with "normal" syntax in direct style, so that both good and bad results of subexpressions unconditionally share the same continuation. Thus, a natural strategy for implementing history accumulation is by either modifying the semantics of function application, or by extending the applied functions with logic for bad value processing. Those are the approaches we use in our Erda implementations, as discussed in section 5.5.2.

Monadic expressions can still be useful for sequencing in Erda, but a mere *identity* monad [Wadler, 1995] is sufficient to achieve the error-monadic effect of not continuing with errors, due to the language semantics. Erda$_{GA}$ includes Haskell-style ***do*** syntax, which is fixed to expand to a "bind" operation that is like that of the identity monad. That operation's definition in Erda$_{GA}$ is simply

```
(define (>>= v f)
  #:alert ([bad-arg pre-unless (function-with-arity? f 1)])
  (f v))
```

As the **>>=** function is not a handler, it does not get called with `Bad` values, which is useful for short-circuiting expression sequences upon an error. For example, the function `compute-a-lot` does not get called in the expression

```
(do [y <- (raise 'bad)]
    (compute-some-more y (compute-a-lot 42)))
```

which is equivalent to

```
((raise 'bad) . >>= . (λ (y)
  (compute-some-more y (compute-a-lot 42))))
```

whose result has a record of the failure of the particular **>>=** application.

5.6 Related Work

IEEE floating point [IEEE, 2008] NaN (not-a-number) value handling is similar to our implicit bad value propagation scheme: with "quiet" (as opposed to "signaling") NaN arguments, most operations produce a NaN result. Thus, badnesses can propagate through a larger computation, without any explicit intermediate checks, leaving the programmer with more choice in choosing a point of recovery. The IEEE standard's default "exception" reporting behavior for invalid operations of a floating point result type is also to produce a quiet NaN. IEEE NaNs are encoded using bit patterns (within raw floating point value data), rather than using a wrapper type, as in our approach; a way to locally switch to IEEE error propagation in Erda$_{GA}$ was demonstrated in section 5.5.3.

Zig[16] is a programming language that has a built-in error type, whose values are globally unique names; error values must be declared prior to use, but they may be declared multiple times. Similarly to our support for bad-value-extended data types, Zig features specific support for "error union types" [Kelley, 2016], whose values are tagged as *either* of a built-in error type, or of some other type `t`. The error union type of `t` can be specified concisely as **`%t`**, and Zig also features specific syntax for conveniently processing values of such types. For example, Zig's binary operator **`%%`** unwraps its left operand where it is a non-error, or evaluates to its right operand otherwise, whereas a **`%return`** expression similarly either unwraps any non-error value of its subexpression, or **`return`**s with the error. Unlike Erda, Zig has no support for implicit error processing, but one could write Erda-style error propagation

[16] `http://ziglang.org/`

code manually by using error-union-typed variables throughout (i.e., also for function arguments).

Rust[17] is a programming language that appears to have influenced Zig, and it also features specific support for reporting errors in terms of sum types. That support expects the use of `Result` types with `Ok` and `Error` value constructors of user-selectable concrete types; i.e., `Result` is roughly like Haskell's `Either` type constructor. Like Erda, Rust features hygienic macros, and uses them for predefining syntax for purposes of error handling; the `try!` macro, for example, has semantics matching Zig's built-in **`%return`** construct. Rust enforces error handling by issuing a warning about unused `Result` values. Rust has no support for implicit error processing.

Applicative functors [McBride and Paterson, 2008] are another categorical structure that has wide applications in programming, although monads are perhaps better known. It is possible to define an "error applicative" that deals in good and bad wrapped values, keeps computing in the face of errors, and also accumulates error history. McBride and Paterson [2008, section 5] have shown a example of such an applicative, one that otherwise behaves like our error history extension for primitive functions, but which collates results into an abstract "monoid" rather than building terms detailing the failed function calls.

Swamy et al. [2011] have presented a rewriting algorithm (for extending the ML language) to insert the necessary glue code to allow programming with monadic types $m\ \tau$ as if they were of the bare type τ. The solution could be used to achieve implicit, pervasive error propagation for a program that is appropriately typed in terms of an error monad, but error monads do not accumulate error history.

Kotelnikov [2014] has extended Scala with similar functionality for type-directed transformation. The macro-based extension is able to generate glue code not only for monads, but also other computation types, making it usable not only with error monads, but also error applicatives, for example. The solution is not language wide, as only `context` expressions are transformed. The desired computation type has to be explicitly specified for each use of the `context` macro.

Kammar et al. [2013] have extended Haskell with constructs for *algebraic effects* [Plotkin and Power, 2001] and *effect handlers* [Plotkin and Pretnar, 2009], with the extension implemented in terms of the Template Haskell [Sheard and Jones, 2002] and quasiquote [Mainland, 2007] facilities of GHC. An algebraic effect signature abstracts over computational effects, while an effect handler modularly specifies a concrete implementation for such an effect interface. As demonstrated by Kammar et al. [2013, section 2.3], an effect handler can specify an interpretation for a "failure" effect, for example in terms of an error monad. GHC's compile-time meta-programming facilities might similarly allow for providing usable syntax for our history-accumulating error processing semantics.

[17] `https://www.rust-lang.org/` (2016)

5.7 Conclusion

We have described a highly portable error reporting and propagation convention in which errors are represented as normal data values, and in which all operations are made to appear *total*[18]; in effect, abnormal control is traded for abnormal data. The convention is made possible by uniformly extending all data types so that their values are distinctly either *good* or *bad*. The convention is founded on the theory of guarded algebras, to provide a formal basis for reasoning about convention-obeying programs' error behavior.

We have shown that the convention is compatible with both explicit and implicit language-integrated support. While errors are just values, and thus manipulatable as such, a language can provide traditional-style "exception" raising and handling syntax dedicated to explicit failure management. A language can also provide "normal" algorithmic language syntax (allowing programming in direct style), while still supporting implicit triggering and propagation of errors as data values, even in a language-wide manner.

Such transparent error processing can be made more capable for semantically rich programs; if any possible error conditions and their reporting mechanisms are declared for each function, then the language can fill in the glue code for either triggering a fresh error, or shielding the function from being exposed to existing badnesses which it cannot handle. Thus, if an API is correctly implemented and annotated, then none of its uses can induce uncontrolled behavior, giving some guarantee of robustness.[19] The automated error processing's freedom from side effects facilitates static program analysis, which can also benefit its supporting language's optimizations.

From a programmer's point of view, pervasive and automatic treatment of errors has the potential to make recovering from failures significantly easier. In particular, inserting error handlers should become easier when ① they interact with the normal control flow of data and not some exceptional control flow, and ② there is no need for explicit checking of error values at every call site.

Acknowledgements

This research has been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language.

[18] A *total* operation is valid for all values that its parameter types could take.

[19] The robustness guarantee does not extend to ensuring that stateful operations are invoked in a correct and *complete* sequence, so that there can be no resource leakages.

CHAPTER 6

Mouldable-Language-Based Niche-Platform Product Lines

> "Vision without action is a daydream.
> Action without vision is a nightmare."
>
> A Japanese proverb

This chapter presents a concrete but unproven vision of how the solutions so far presented might be combined to build a niche platform software product line, one that adheres to section 1.7's product-line strategy. The architecture presented here is *speculative*, and while parts of the technology for it have been implemented, the overall design has not been validated. I also fully expect that some of the technical details sketched in this chapter would turn out to be poor design, or even outright unimplementable as described. My aim with this exposition is to give an idea of how the individual technologies *might* fit together, which also then sheds light on the motivations behind those technologies, since they were created with this kind of use case in mind.

As a basis for the presented product-line architecture, a mouldable programming language is required. With Magnolia and Magnolisp as starting points, I could go in either direction to build a hybrid language that combines some of the features of both languages for more fully realized mouldability.[1] As the implementor of Magnolisp, however, I am better prepared to speculate on what would be required to make Magnolisp sufficiently mouldable, and so I have chosen to use a hypothetical, more advanced Magnolisp as a language-technological basis for this chapter's product-line architecture.

For the remainder of this chapter, I shall refer to the existing Magnolisp technology as $Magnolisp^{v0}$ (for "version 0"), with the imagined Magnolisp technology being assumed otherwise. I may in places use the superscript "v1" to emphasize that I am referring to the latter.

[1] Bagge has previously explored the direction of having Magnolia support variability [Bagge, 2010a] and extensibility [Bagge, 2010b], but those mechanisms are not in use in the present Magnolia.

6.1 The Magnolisp Language Family and Infrastructure

The language infrastructure of Magnolisp^{v1} is called *Magnolisp*$_*$, and features not only a macro system, but also a configurable `mglc` compiler, allowing it to support domain-oriented flavors of the language that also differ in their core language. *Magnolisp*$_{\texttt{lang}}$ shall denote any language implemented on top of Magnolisp$_*$, *Magnolisp*$_{\texttt{base}}$ shall denote any language featuring the default core language of *Magnolisp*$_{\texttt{,base}}$, and *Magnolisp*$_{\texttt{alt}}$ shall denote any language featuring some alternative (and incompatible) core language.

The central Magnolisp$_{\texttt{base}}$ language is *Magnolisp*$_r$ (for "reasoning" or "restricted"), or just "Magnolisp." That Magnolisp is similar to the current Magnolisp^{v0} language, particularly with respect to its static reasoning friendly core language; however, it is also intended as a product line's language for software composition, and has for that reason been augmented with a Magnolia-influenced component system featuring concepts and external linking. Magnolisp$_r$ is *the* mouldable programming language of this chapter's PLA design, but it has a close relative in Magnolisp$_{Erda}$, which may generally be used as an alternative to Magnolisp$_r$. Magnolisp$_u$, Magnolisp$_{C++}$, Magnolisp$_{Qt}$, etc. are Magnolisp$_{\texttt{alt}}$ flavors, intended for implementing types and their operations for Magnolisp$_{\texttt{base}}$ languages, rather than for implementing entire programs.

Magnolisps are generally all alike in a number of ways (although exceptions can be made to suit the domain orientation of a given language flavor):

- They are implemented as a Racket `#lang`.
- They have a Racket-themed surface syntax.
- Their name resolution is compatible with Racket's.
- They use Racket's module system natively for namespace management.
- Their foreign language interface descriptions are exported via submodules.
- They all expose Racket's macro system for self-extension, and use it as a platform for reusable language features, as suggested in section 3.10.1.

Magnolisp$_{\texttt{lang}}{}^{v1}$ languages may have a slightly different appearance compared to Magnolisp^{v0}, as they feature additional shorthand annotation syntax; for example, `#:: ([type Int])` may alternatively be written as `#:type Int`.

6.2 A Product-Line Architecture

This chapter's product-line architecture supports product configuration, software composition, and building, as outlined in figure 6.1. In more detail, the technologies used to manage those high-level aspects of the software production are:

- Konffaa is used as a configuration manager, with its Racket-based `konffaa` language used as a variant specification language. One variant must be selected before software composition or building, by invoking `konffaa` on the desired variant specification; this is similar to invoking a GNU Autotools based `configure` script before building, except that any configuration options come from the selected variant specification file, rather than being specified through command-line arguments.

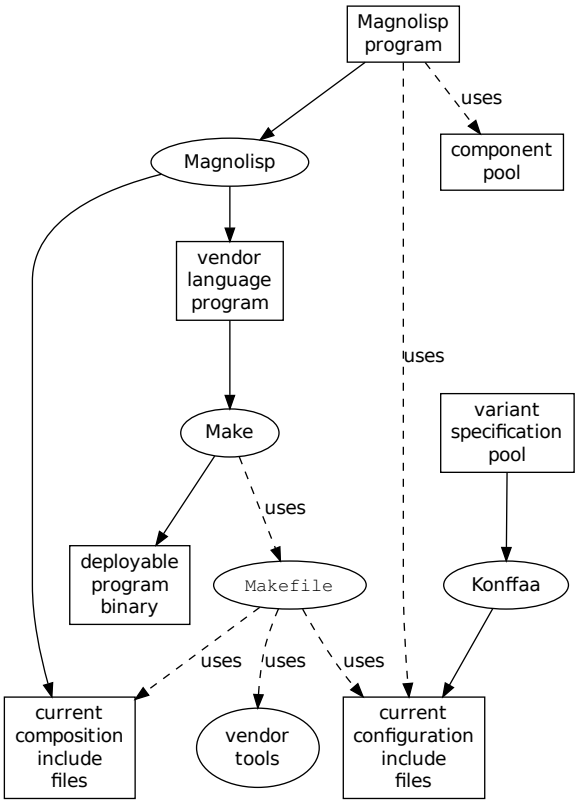


Figure 6.1: A product-line architecture overview, showing configuration, API, and build management at a high level. The component pool serves as an API knowledge repository. The collection of variant specification files serves as a configuration knowledge repository. The `"Makefile"` and vendor tools in turn encode build knowledge. The `konffaa` and `make` CLI tools are used to trigger configuration and build actions, whereas simulation of the program may be driven through Racket and the `magnolisp` language.

- Magnolisp$_r$ is used for composing a program for each variant. The `magnolisp` source file exporting the program can be named in the variant specification, if there is more than alternative program source file. A program-composition-expressing source file may have a compile-time dependency on the variant specification to allow for conditional compilation, which may be sufficient for expressing the necessary composition variability.
- GNU Make is used as a high-level build manager, and its language is thus used to express configuration-dependent details about how to invoke other tools for building and packaging the program. The makefile describing those details can depend both on Konffaa and Magnolisp output, describing both configuration parameters and inferred details about the composition. Other tools invoked by the makefile might include a vendor SDK's build manager, in which case a suitable build description may have to be generated as input for that lower-level build manager.

The product line requires a pool of core assets, and these can be implemented in a variety of languages:

- The `konffaa` language is used to specify parameters for configurations. Additionally, any vendor-supplied project managers' languages may be used to specify target-specific details.
- GNU Make's language is used to encode build instructions for the relevant combinations of components and targets. Additionally, any vendor-supplied build managers' languages may be used to specify target-specific details.
- Magnolisp$_{lang}$ languages and vendor programming languages may be used to implement software components. Any component implemented in a target language must be given a Magnolisp declaration, to enable its management by Magnolisp; Magnolisp$_{alt}$ languages automatically generate a Magnolisp declaration for their Magnolisp-facing exports.
- Racket (and its `#lang` mechanism) can be used to implement further Magnolisp$_{lang}$ languages as libraries.
- Any resource files (e.g., application or user interface description files) are specified in vendor-specific languages. Alternatively, macro-defined constructs might generate any auxiliary resources they require, as a side effect during macro expansion.

6.3 Managing Configurations with Konffaa

To make it easier for humans to refer to interesting product variants, it is useful to give each one of them a descriptive name. The Konffaa configuration manager—introduced in section 1.3.1—allows for this, also featuring command-line completion of variant names when invoking the *`konffaa`* CLI tool. Each variant *`name`* maps to a "*`name`*`.var.rkt`" file, expected to be written in the `konffaa` Racket-based language, and to state the defining configuration parameter values for that variant.

Before building a product, one variant must first be selected by invoking `konffaa` with a variant as an argument, as illustrated in figure 6.2. Assuming the selected variant's axioms hold, the invocation results in a set of include

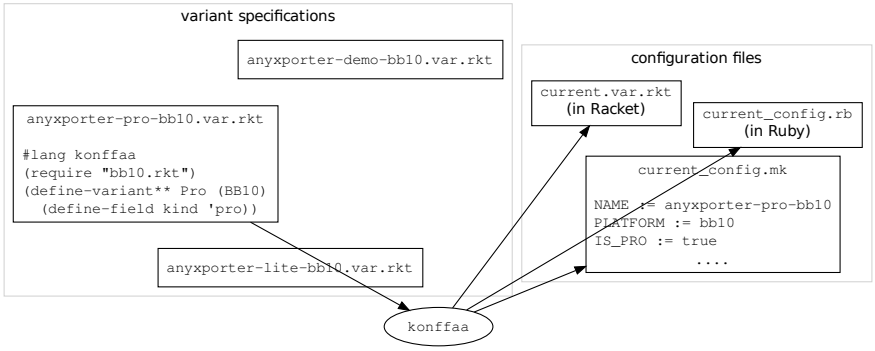


Figure 6.2: Configuring the Anyxporter codebase for its "Pro" BlackBerry 10 variant with the `konffaa` tool. Only the GNU Make, Racket, and Ruby include file formats are included in the diagram.

files being generated in various languages. The include files can be used for conditional macro expansion in Magnolisp*lang* languages, conditional compilation in vendor languages (e.g., where they feature the C preprocessor), or for specifying conditional build instructions.

6.4 A Macro-Implemented Component System

As we discussed in section 2.4.2, macros can be used to implement component systems. Pattern-based macros in particular are commonly used to compose and parameterize code statically, and thus they can already serve a similar purpose as they are. However, for a cross-platform, domain-engineered codebase it seems worthwhile to standardize on a more structured notion of a program implementation fragment, one supporting separate interfaces.

Towards that end, Magnolisp$_r$ features a macro-implemented, Magnolia-inspired component system with static linking. The component system and language is implemented as a library, and could be reused by other Magnolisps where useful, provided that their run-time core language meets a bare minimum of compatibility requirements. Like Magnolia, the component language features concepts, axioms, and satisfactions, but the language otherwise resembles that of Racket's units [Culpepper et al., 2005; Flatt and Felleisen, 1998; Flatt and PLT, 2010], or rather the static information managing `define-` and `/infer` subset of it. Magnolisps' component naming convention also follows that of units, in that concept and component names are suffixed by `^` and `@`, respectively.

Following terminology by Flatt and Felleisen [1998], I say that a Magnolisp component is *linked* with others to form a compound component. I say that a Magnolisp component is *invoked* when its code is subjected to compile-time evaluation (or macro expansion), and its exported bindings are made available to a particular local or top-level context.

A component language need not be limited to defining, linking, and in-

voking components. Goguen [1984, 1986] has advocated having a language of *module expressions* that supports a rich variety of methods for program construction, and in particular allows for combinations of semantically well-behaved component transformations to be specified in a structured manner.

A macro-based component system implementation provides opportunities for extending the component language with new syntax, particularly if the implementation includes a known API for accessing components at the meta level. While it is possible to express higher-level component constructs in terms of more primitive ones, as has been demonstrated by Waddell and Dybvig [1999], an API for accessing component representations makes it possible to implement new component language primitives as well.

Roughly, an atomic component can be represented as unevaluated code for its definitions, and information about its imports and exports. A compound component, in turn, can be represented as a set of its constituent components, and information about the overall component's imports and exports. More detailed examples of compile-time component representations in a Racket-macro-based implementation have been given by Culpepper et al. [2005, section 4.3].

Racket provides several building blocks that a component system implementation for Magnolisp might use:

- Racket supports *module contexts* and *internal-definition contexts* that mingle binding forms and expressions [Flatt, 2015], and which also support mutual recursion for their definitions. Such a context is suitable both for defining components and invoking compositions thereof in order to bring their bindings into the context.
- *General compile-time bindings* [Flatt et al., 2012] are suitable for storing information about components by their name, in a lexical scope respecting way. Like macros, such bindings may be defined with **`define-syntax`**, and the bindings' values are computed at compile time, but unlike macros, their values need not be syntax transformation functions.
- Syntax-quoting provides a way to store code in compile-time state without expanding it, but with its lexical context intact. This makes it possible to delay the (compile-time) evaluation of code containing uses of unbound (or late-bound) identifiers of requires interfaces, while still maintaining referential transparency for other identifiers. Quoting as syntax can be done for instance with the ***`quote-syntax`*** core form, or the derived `#'` shorthand syntax.
- A *rename transformer* is a special macro that substitutes an identifier for another, and such a macro can translate between component internal and external names. Firstly, upon static parameterization of a component, a renaming may be defined to introduce a mapping for a late-bound parameter, to translate its uses to those of an externally-specified, already-bound identifier. Secondly, a renaming may be defined to expose a binding (under a desired name) that maps to a component-defined, internal binding, as part of invoking a fully-implemented component in a particular context.
- Racket's facilities (such as the **`make-syntax-introducer`** function) for manipulating identifier scopes may be used to explicitly control scoping,

for instance to ensure that no clashes between components' internal names can occur as their code is being multiply instantiated for the same context.

- Sub-form expansion and analysis (for example in terms of **`local-expand`** and **`syntax-case`**) may be used to discover bindings appearing at a component's top level, even when late-bound identifiers appear on the right-hand sides of the definitions for the bindings. This makes it possible to use macros within a component body, while still being able to examine (perhaps partially) expanded components to determine what bindings they contain, perhaps to infer or check the component's exports.

The above facilities are all suitable for implementing component language that is fully static, such that all implementations are resolved statically; in particular, components need not be run-time values that merely implement a statically known interface. The simple `define<>` and `use` example of figure 2.3 already demonstrated the first four of the six facilities.

It is quite common in C and C++ to define macros in API-describing header files, and Magnolisp (and Racket) can similarly export macros from modules [Flatt, 2002], as abstractions over syntax. It is also possible to allow *component* interfaces to contain macro definitions, as has been shown for the combination of Racket units and macros by Culpepper et al. [2005], and Magnolisp$_r$ does allow concepts to contain definitions of syntax extensions. Furthermore, unlike in the case of units, in Magnolisp all component uses are known statically, which also makes it possible to allow its component *implementations* to import and export macros.

6.4.1 Foreign Component Interfaces

In this product-line arrangement, an important motivation for having a component system is to put APIs over target-specific implementations. This in turn makes it important to have a sufficiently capable *foreign function interface* (FFI) mechanism to allow target languages to implement those APIs, preferably in a way that retains the genericity aspect of mouldability.

Magnolisp^{v0} can already declare foreign types and functions, but the Magnolisp^{v1} component system would benefit from the ability to statically parameterize foreign APIs, whose implementations are opaque to Magnolisp$_r$. As the available mechanisms for such parameterization vary between target languages, any solutions should probably vary as well, to ensure that it is reasonably convenient to craft a foreign API according to the FFI's conventions. The chosen FFI convention might not only depend on the target language, but perhaps also on the nature of the API, to best facilitate lightweight foreign implementations.

Parametric polymorphism is a form of generics that is well supported by a number of potential target languages, and thus any FFI mechanisms achievable in terms of generic foreign types and functions should be sharable across targets, meaning that separate Magnolisp declarations might not be required for implementations in different languages. The idea is simply that generic types and functions should be able to adapt to whichever types we ultimately configure for our components, and that the type context of the uses will tell the target language what it needs to know about our configuration. For

example, in these Magnolisp definitions we assume that the foreign entities `Box-t`, **`box`**, and **`unbox`** will adapt to the chosen box element type `BoxE`:

```
(define-concept Box^
  (typedef/abstract Box) ; abstract box type
  (typedef/abstract BoxE) ; abstract element type
  (define/abstract (box e) #:type (-> BoxE Box))
  (define/abstract (unbox b) #:type (-> Box BoxE)))

(define-component Box@
  #:requires ([BoxE #:from Box^]) ; BoxE's signature is as in Box^
  #:provides (Box^) ; all in Box^ (excluding any already required BoxE)

  (typedef/foreign Box-t) ; foreign type named Box_t

  (typedef Box (<> Box-t BoxE)) ; type alias for Box_t<BoxE>

  ; foreign functions, with the rkt. expressions specifying simulation
  (define/foreign box rkt.box-immutable)
  (define/foreign unbox rkt.unbox))

(define-satisfaction Box@-is-Box^
  ; Box@ implements Box^ (with any type BoxE)
  [Box@ #:models Box^])
```

For truly general support, we must be able to parameterize both types and functions, but the notion of a component does not necessarily need to exist on the foreign language side; it is up to the component system to ensure consistent use of foreign APIs in relation to a component. After any component language has been macroexpanded away, the remaining uses of foreign entities will be limited to specific abstract types and functions. Those uses may have to be parameterized, and for Magnolisp it should be sufficient if foreign types can have types as static parameters, and if foreign functions can have either types or functions as static parameters. For a target language that supports such types and functions, it should be possible to define an FFI for Magnolisp such that it allows for the full Magnolisp component language to be used on target-language-implemented components, and such that component linking happens statically at the level of target language source code.

In the C++ language, static configuration of data structures and algorithms is possible by specifying concrete types and functions as template arguments. As an example, we might consider a variant of the `modify-first!` function of section 1.4.2, one for which the replacement-value-computing function is specified statically rather than dynamically. We might specify that computation as a "functor" type (i.e., one that implements `operator()`), and we might allow any `Collection` type for which `get_first` and `set_first` is implemented (in terms of ad-hoc polymorphism). We can leave it to the functor named by the `Modify` template parameter to determine the supported element types:

```
template <typename Coll, typename Modify>
void modify_first(Coll& coll) {
  set_first(coll, Modify()(get_first(coll)));
```

```
}
```

Armed with such parameterization facilities in the target language, what remains is to ensure that Magnolisp's FFI allows passing of those parameters, and that the component system can keep track of which parameters to pass.

Target languages lacking the static expressiveness of C++, on the other hand, may require alternative FFI solutions to work around their limitations. One might, for example, accept some dynamic overhead in component composition; mbeddr does so in its default configuration of connecting components via function pointers in its generated C code [Voelter et al., 2013a].

Another alternative is to bring the target languages under Magnolisp$_*$ control, by hosting them as S-expression-based macro-enabled target-language reformulations (roughly in the style of C-Mera), to make their code less opaque to the Magnolisp$_*$ environment than the ultimate target languages are. This may enable the use of the Magnolisp component system itself to parameterize the not-so-foreign code, and in that way instantiate the required specializations of that code, translatable quite directly for the actual target. Target-specialized flavors of Magnolisp are discussed further in section 6.9.

6.5 Composing Programs in Magnolisp$_r$

While components may be implemented in a variety of languages, it is Magnolisp$_r$'s role in the product-line architecture to manage the components, express program compositions, and prepare them for targets, as illustrated in figure 6.3.

To specify the different compositions of a line of products, there might be just one Magnolisp program that refers to different configuration parameters in creating an appropriate composition for each product variant. For example, suppose we wanted to compose a compass application for both bada and Tizen, with the APIs, components, and desired compositions illustrated in figure 6.4. To express the compositions in code, we might use section 1.4.6's `static-cond` to select the appropriate sensor component for the target platform, referring to the configuration parameters in `"config.rkt"`:

```
(require (for-syntax "config.rkt")) ; as generated by Konffaa

(define-compound CompassApp@
  #:provides (run) ; run's signature is known from Engine@
  #:link ; together three components
  ((static-cond
     [is-bada? BadaMagneticSensor@]
     [is-tizen? TizenMagneticSensor@])
   OpenGlCompassView@ ; assume OpenGL ES for all targets
   Engine@)) ; platform-agnostic application engine

(define (run-main) #:export ; program entry point
  (invoke-component CompassApp@) ; bring run into scope
  (run))
```

With a particular configuration chosen, `raco make` will take care of rebuilding Racket bytecode as required, to match the configuration; if `"config.rkt"`

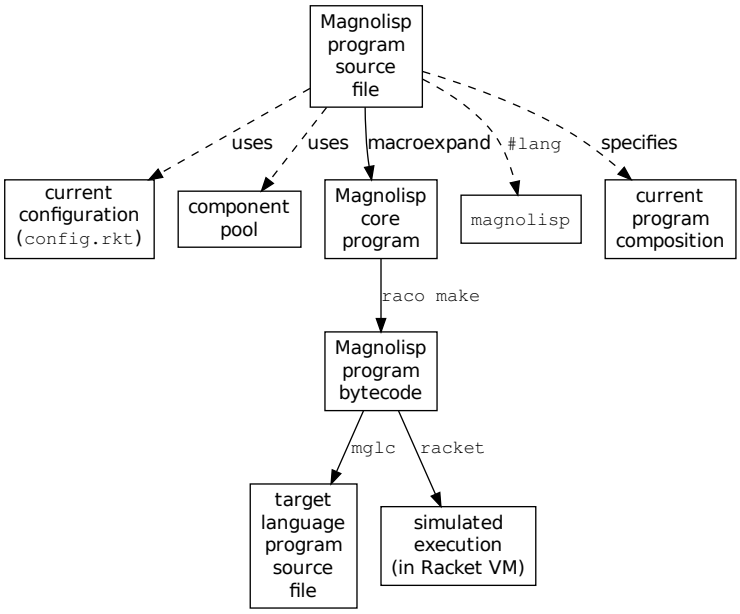


Figure 6.3: The composition of a product variant's application logic may be expressed in `magnolisp`'s component language. Linking of components may be conditional on the Konffaa-generated `"config.rkt"` file that lists the configuration parameters of the variant. A program composition may be further processed to produce a target-language equivalent, or executed in the Racket VM (to the extent that simulation behavior has been specified for external implementations).

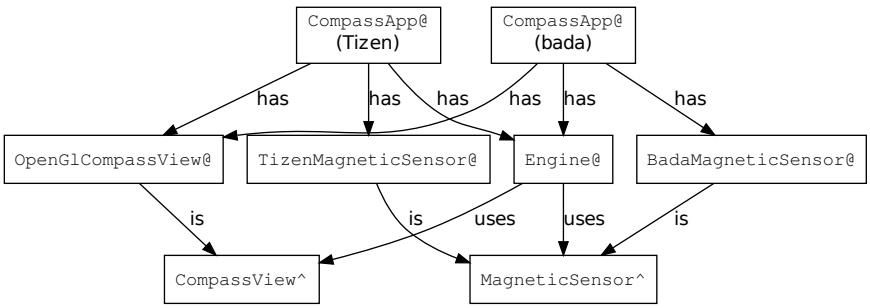


Figure 6.4: bada and Tizen compass application compositions, whose platform-agnostic `Engine@` refers to sensor reading and view rendering functionality only via the abstract `MagneticSensor^` and `CompassView^` APIs.

gets replaced by a different configuration, `raco make` should detect the change. It is not necessary to switch configurations for purposes of simulation, if the selected configuration's foreign operations have Racket implementations. To the extent that they do, the program's operations (or axioms) can readily be invoked in a "simulated mode" within the Racket VM.

A Magnolisp component's `#:requires` and `#:provides` interfaces are specified explicitly, which should help the programmer keep track of them, even without specific IDE support for inspecting compositions. Like Racket's units, Magnolisp's components distinguish between external and internal names [Flatt and Felleisen, 1998]. In Magnolisp, a correspondence between the two kinds of names is established in terms of `#:requires` and `#:provides` clauses, which allow the renaming of individual imports or exports `#:as` something else, or assigning a `#:prefix` to all imported or exported names. Specifying renamings with the same clauses is also possible for component *invocations*, but in that case the mapping is between component-external and invocation-context-local names. Requires and provides are not applicable to concepts and satisfactions, which are defined solely in terms of external names.

Component bodies' internal definitions stay internally scoped unless explicitly exported with `#:provides`. Any externally sourced definition should be declared with `#:requires`, or its name should be bound in an outer scope to a definition not involving dynamic extent (typically a top-level type or function). Component linking by default works by matching external names, and renaming may be used to get names to match as desired. Linking in terms of unit-style "tags" [Culpepper et al., 2005] is also supported, for scenarios in which name-based linking becomes unwieldy. Within a `#:requires`, it is possible to reference a signature `#:from` an existing concept, to avoid having to re-declare it; this is unnecessary where the connection is obvious from context.

Magnolisp's component language constructs include:

- `define-concept`: Defines a named concept, whose interface should only contain abstract types, abstract functions, and macros. A concept may implement axioms, which are not a part of the interface.
- `define-component`: Defines an atomic component. A component may have both `#:requires` and `#:provides`.
- `define-compound`: Defines a compound component. Unless explicitly specified, `#:requires` its components' internally unsatisfied imports. Unless explicitly specified, `#:provides` its components' exports that are not linked internally.
- `define-satisfaction`: Defines a named claim stating that the specified component satisfies (or `#:models`) the specified concept. This establishes a connection between a concept and one or more of its implementations; unlike in Magnolia, doing so may effectively enlarge the interface of the implementation, since a concept's interface may include macro definitions.
- `define-axiom`: Defines a named axiom.
- `invoke-component`: Invokes a component by its name. Unless otherwise specified, uses the local context's bindings of the same name to satisfy any `#:requires`. Unless otherwise specified, binds any `#:provides` locally under their external names, excluding any definitions that the

same invocation `#:requires` under the same external name (since they then are already bound under that name).

Magnolisp's components can be used in the small, as a substitute for higher-order functions (with a different parameterization time). For example, we might create an anonymous component to specialize section 6.4.1's `Box@`, and immediately invoke it for a local context in order to bring its export bindings into scope:

```
(define (Int-box v) ; (-> Int (<> Box-t Int))
  (invoke-component Box@ ; binds all names of Box^ locally
    #:requires ([Int #:as BoxE]))
  (box v))
```

Should some of the above syntax seem verbose for what it does, the advantage of Magnolisp is that shorthand syntax can readily be defined for use cases that turn out to be common, even if that use case is only common in a particular product line.

For Magnolia's built-in component language, there are plans to introduce constructs for the domain of algebraic specification. The planned constructs are for defining **`congruence`**s based on predicates (and inducing the related axioms), and for deriving component implementations for congruence-induced **`quotient`** constructions, for example. Ideally, Magnolisp$_r$ would be able to support the implementation of such constructs as macros.

There are limits to what can conveniently be implemented merely based on macro-captured information, without separate static analyses phases for inferring further information. Macros can certainly also perform static analysis, as demonstrated by Typed Racket [Tobin-Hochstadt et al., 2011], but some features are perhaps most practically implemented in collaboration with the Magnolisp compiler. Where custom core language is added to support a feature, however, and Racket VM execution is also required for that feature, one should be mindful that the code for Racket execution may then also require translation for the added core language (as mentioned in section 2.3.2).

6.5.1 Using Program Compositions

Magnolia allows a component to be designated as a **`program`**, which results in the generation of a CLI for accessing its available operations. As Magnolisp is intended to support a variety of execution environments, it makes few assumptions about how a Magnolisp program will be launched. It is up to the programmer to decide which "entry point" function (or functions) to export to the target language for purposes of running a Magnolisp program. It is furthermore up to the programmer to write any C++ `main` function for invoking the Magnolisp entry point from the command line, for example, or whatever else may be required for the particular product variant.

If one wants to handcraft an application's user interface for a platform-faithful look-and-feel, a Magnolisp "program" might actually just be a library of application logic, to be used from hand-coded user-interface code. In the case of Qt, for example, one might expose QML[2] bindings to a C++ translation of that logic, for purposes of programming a user interface.

[2] *QML* is Qt's domain-specific UI "markup" language.

Magnolisp as an extra language layer with knowledge of domain-engineered APIs provides opportunities for automating the generation of foreign-language bindings. In addition to exposing a target language interface (e.g., for Racket using **`provide`**, or for C++ using `#:export`) for generated code, Magnolisp might be made to additionally generate QML or Lua bindings for specially annotated APIs, for example, should that be useful in a given product line.

To assist in implementing test configurations, Magnolisp's component language might include an operator for producing a routine for invoking a component and running all of its associated axioms, with semi-randomly generated data as arguments. Such routines could then be used to implement an entry point for a test program.

Another example of potentially useful component language is mbeddr's **`mock component`** [Voelter, 2014] construct for turning a declarative specification of behavior in a specific *scenario* into an component that implements that behavior, but only for when the scenario unfolds as declared (i.e., the expected sequence of calls is made on the component).

6.6 Cross-Component Error Handling in Magnolisp$_{Erda}$

Magnolisp$_{Erda}$ is a language derived from Magnolisp$_r$, one that features implicit propagation of error information in data values, as described for Erda in chapter 5. Magnolisp$_{Erda}$ has failed-expression-history recording disabled by default, but does provide explicit control over where recording happens, along the lines suggested in section 5.5.4. History **`redo`**ing is not supported, however, to ensure portability, and to avoid inconsistencies between original and replay execution in cases where side effecting operations are used.

Magnolisp$_{Erda}$'s error propagation is implemented in terms of macros, and like Magnolisp$_r$, the language uses Magnolisp`,base` as its core. Magnolisp`,base` includes specific language for error propagation to allow `mglc` to optimize in an error-aware manner and to emit more semantically rich output. Magnolisp$_{Erda}$ may nonetheless transcompile to more cluttered code than Magnolisp$_r$, which is the tradeoff to be made in choosing between `#lang magnolisp` versus `magnolisp/erda`.

If desired, it is possible to allow that tradeoff to be made for individual definitions or code fragments. For example, section 5.5.3 showed Erda$_{GA}$'s **`let-direct`** expression for switching off implicit error processing for a code block. Erda$_{GA}$ also features a ***`define-direct`*** form for defining an entire function with implicit processing disabled, so that section 5.5.3's example function `f` can be written more concisely as

```
(define-direct (f a b x)
  #:alert ([NaN post-when (nan? value)])
  (fl* (fl+ 1.0 (fl/ a b)) x))
```

Since Magnolisp$_r$ and Magnolisp$_{Erda}$ share the same core, it should be quite possible to adopt the above idea for both of the two languages to make them duals of each other with respect to their error processing behavior. That is, Magnolisp$_r$ could be the "direct-mode" language for Magnolisp$_{Erda}$, and Magnolisp$_{Erda}$ could be the "Erda-mode" language for Magnolisp$_r$, with forms

such as **let-direct** and let-erda provided for locally switching between the two modes.

It is also possible to adhere to Magnolisp$_{Erda}$'s error reporting convention without any specific language support, thus allowing it to serve as a cross-language convention, as desired. Some Magnolisp variants might opt to use the convention implicitly for error reporting at their API level, or even for their internal error propagation. Other variants might support the convention more explicitly (e.g., in the style of Zig), thus leaving the programmer in control over the generated error processing code.

The readability of Magnolisp$_{base}$ languages' C++ translation is enhanced through mglc's use of target-language-defined abstractions relating to error processing, in order to reduce clutter and to make semantics clearer. For where error checks remain, the back end emits MGL_LIKELY and MGL_UNLIKELY macro uses as branch prediction hints, indicating the non-error case as likely; these macros can be implemented for example in terms of GCC's __builtin_expect function, where available.

6.6.1 Resource Cleanup

A concern closely related to error handling is that of resource cleanup. To allow the cleanup action for an allocated resource to be specified in the syntactic proximity of the allocation expression, Magnolisp$_{Erda}$ features begin-defer and defer constructs. Their use is illustrated by the following **copy-file** routine, whose "plain" version appeared in figure 5.1b. The deferred expressions essentially get moved to the point of exit from the enclosing begin-defer block, and there is automatic bookkeeping to keep track of the defer expressions that are actually reached during the evaluation of the block:

```
(define (copy-file from to)
  (begin-defer
    (define in (open-input-file from))
    (defer (close-input-port in))
    (define out
      (if (good-result? in)
          (open-output-file to)
          (raise 'undefined)))
    (defer (close-output-port out))
    (copy-port in out)))
```

Magnolisp$_{Erda}$'s defer construct also supports an #:on-error modifier to indicate clean-up that should *only* happen when the begin-defer block exits with a bad value, to allow a successfully initialized resource to be preserved for later use. The results of deferred actions themselves (whether good or bad) are ignored. Thus, a plain defer is like the D[3] language's **scope(exit)** statement or Zig's **defer**, whereas defer #:on-error is like D's **scope(failure)** or Zig's **%defer** [Çehreli, 2016; Kelley, 2016].

The begin-defer and defer constructs are implemented as macros, with each appearance of defer replaced with an assignment to a fresh bookkeeping variable, to indicate that it was reached. The primary restriction in specifying

[3] http://dlang.org/

deferred actions is that they may not use variables that are not in scope in the internal-definition context of the enclosing begin-defer.

A resource-cleanup abstraction of this nature has general appeal, and the bookkeeping strategy is also generally applicable, suggesting that these macros might be worth adapting to some other flavors of Magnolisp as well. In doing so one would have to account for any non-local control transfers in those languages. Furthermore, the #:on-error syntax would probably have to be excluded for flavors that lack a convention for error handling.

6.7 A #lang Configurable mglc

Magnolisp has a conservative, constrained language design in order to better support reasoning about program compositions. For purposes other than expressing program compositions, it may be desirable to have flavors of Magnolisp that make different design tradeoffs, for example trading static reasoning ability for dynamism. As such design tradeoffs would tend to also impact core language, Magnolisp$_*$ includes a mechanism for allowing a #lang to control its compilation, so that it can instruct the compiler to make compatible assumptions about expected core language.

Magnolisp^{v0}'s channel for communicating information to mglc is through a submodule, as described in section 2.3.5. Magnolisp^{v0} already configures mglc by specifying the desired "runtime" libraries (which declare information about the actual C++ runtime APIs) through the magnolisp-s2s submodule:

```
(module magnolisp-s2s racket/base
  (#%module-begin
    ....
    (define-values (prelude-lst) '(magnolisp/2014/prelude))
    ....
    (#%provide .... prelude-lst ....)))
```

The mglc^{v1} compiler uses the same channel for its options, also accepting options relating to core language.[4] In particular, magnolisp-s2s defines an alt-core? boolean variable, which may be set to indicate that the module is implemented in a different core language. If that is the case, then there is also expected to be a magnolisp-alt submodule supplying the implementation, and specifying its core language[5]; magnolisp-s2s then merely declares the Magnolisp$_{'base}$ interface, so that Magnolisp$_r$ can still serve in its software composition role. Figure 6.5 illustrates the submodule communication of an example Magnolisp$_{alt}$ module.

To specify a Magnolisp$_{'base}$ interface for a Magnolisp$_{alt}$ module, its constituting types and functions must be flagged as *#:magnolisp* to make them available to mglc, and also **provide**d to make them available to Racket. The

[4] As the options are passed on a per-module basis, this strategy even makes it possible for core language adjustments to be dependent not only on a module's implementation language, but also on its content.

[5] To go beyond selecting from a fixed set of core language choices (as assumed in this chapter), the alt-core language might be specified as a Racket module implementing its compilation, thus giving each #lang open-ended flexibility in controlling its transcompilation.

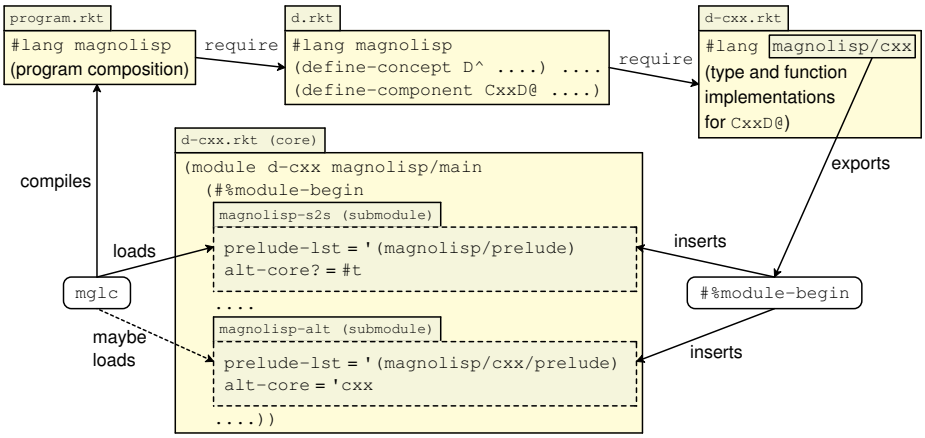


Figure 6.5: A "d-cxx.rkt" module written in the magnolisp/cxx language. The language has its own runtime library, and requests the Magnolisp,cxx core language from mglc. When mglc compiles a Magnolisp "program.rkt" that **require**s "d-cxx.rkt" (via the "d" component definitions in "d.rkt"), it need not load magnolisp-alt unless "d-cxx.rkt"'s Magnolisp API actually ends up being used. (The name "d" comes from figure 6.6.)

effect of the #:magnolisp flag is to record the flagged definitions' declarations in the magnolisp-s2s submodule in the Magnolisp core language and IR format, but marked as having an alternative-core implementation; mglc can access magnolisp-s2s information as necessary when compiling a Magnolisp program, with the guarantee that the same core language is used throughout.

Each #:magnolisp export may require further annotations for any information that Magnolisp,base compilation requires for static analysis, but which cannot be inferred due to the nature of the Magnolisp$_{alt}$ language. For example, for Magnolisp,base code we can infer build dependencies based on uses of types and functions, but we might not want to attempt that with a Magnolisp$_{alt}$, depending on its characteristics.

Transcompilation of Magnolisp$_{alt}$ modules is driven by the compilation of Magnolisp$_{base}$ code, which actually determines the program composition, and thus the inclusion of Magnolisp$_{alt}$ modules. Only a single target language based implementation file gets generated for all the Magnolisp$_{lang}$ language of a program, which simplifies building, and opens up possibilities for Magnolisp variants to interface with each other within mglc. However, Magnolisp$_{alt}$ modules are not processed together with Magnolisp,base IR all the way through a compilation pipeline, due to the differing core language; alternative core language processing happens initially separately, already due to Racket's separate compilation into bytecode.

To compile a Magnolisp$_{alt}$ module, mglc first loads its magnolisp-alt submodule, which contains the implementation's IR, and specifies the appropriate compilation configuration for it. The configuration must be compatible with the overall compilation pipeline, so that the linking of Magnolisp$_{base}$

and Magnolisp$_{alt}$ code becomes possible at some point during compilation. Linking should at the latest become possible when all the code has been transformed into the target language IR.

A Magnolisp$_{alt}$'s compilation phases are determined by the combination of its core language and the requested target language. Like the Haxe source-to-source compiler, for instance, Magnolisp$_*$ supports multiple compilation target languages (not only Racket and C++). The desired target is specified through CLI or API options, and not by a `#lang`; the set of supported targets is affected by the used core language, as not all Magnolisp$_{alt}$ cores support all targets, and an attempt to use an incompatible combination causes an error.

As Magnolisp$_*$ supports both different source (core) languages and different target languages, its compiler's internals aim for generality and reusability in the design of its IR data structures and compilation passes, to allow for some variability in the languages being processed. Its implementation builds on technologies like chapter 3's Illusyn for data abstraction, and our PGF [Bagge and Hasu, 2013] code formatting pipeline and its pluggable token processors for algorithm reuse.

6.8 More Dynamic Portable Programming in Magnolisp$_u$

The ability to configure `mglc`'s core language enables us to create fundamentally (rather than superficially) different languages, while still building on the Magnolisp$_*$ infrastructure. Haxe, Oxygene, and STELLA are all languages that support transcompilation into multiple target languages, while still having a richer and more "dynamic" algorithmic language than what is enabled by Magnolisp$_{\texttt{,base}}$. Those three languages serve as inspiration for Magnolisp$_u$, whose role in this chapter's PLA is limited to compensating for any restrictions of Magnolisp proper, still from within the same language family and environment, and without sacrificing code portability.

Magnolisp$_u$ (for "unrestricted") is an example of an Magnolisp$_{alt}$ language, one that retains Magnolisp$_r$'s target agnosticity, but trades some reasoning ability for the ability to have more dynamically determined behavior. Magnolisp$_u$ resembles a contemporary object-oriented language, featuring objects with dynamic method dispatch, classes with subtyping, and only local type inference [Pierce and Turner, 2000]; it also features first-class and anonymous functions. A language of such contemporary nature can be compiled to human-readable code for mainstream targets such as C++ and Java, and `mglc` supports those targets (among others) for Magnolisp$_u$.

6.9 Integrating with Targets in Magnolisp$_{C++}$ et al.

We can program component implementations portably in Magnolisp$_r$ or Magnolisp$_u$, but their ability to express target language concepts and to interface with target language APIs is limited. At the other end of the interfacing-ability scale we can directly use the target platform's languages, for full compatibility with the platform. In between these choices, however, there may be room for "bridge" languages designed to interface with both Magnolisp and a specific target language. This kind of a multi-language scenario is sketched in figure 6.6.

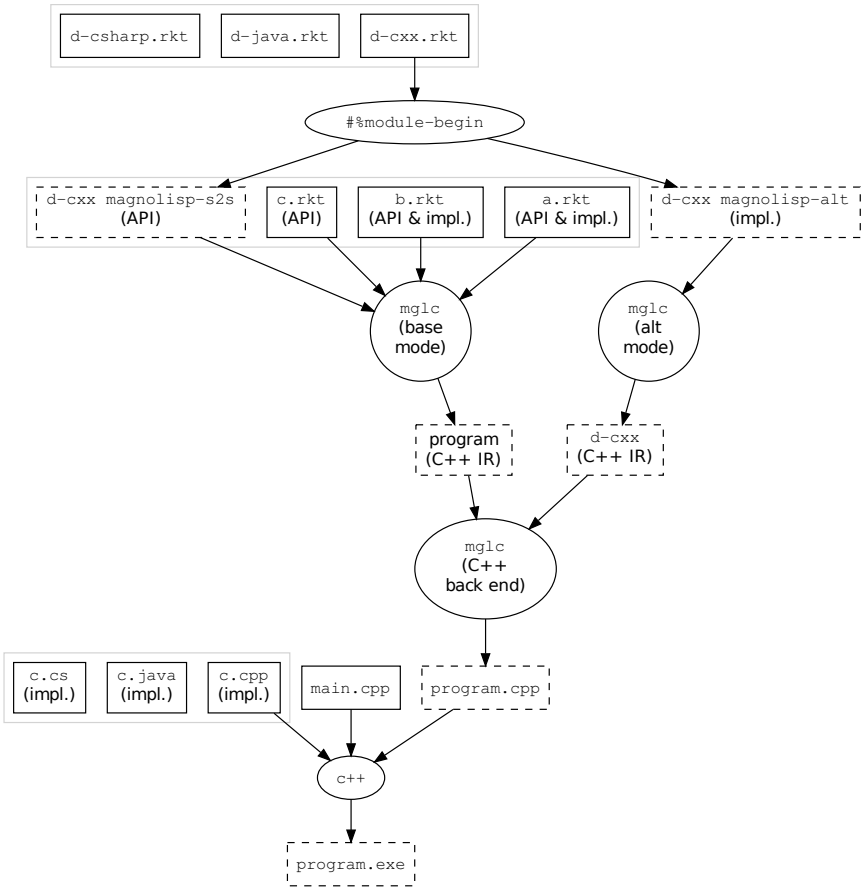


Figure 6.6: Assembling and compiling an executable program from a selection of components in multiple languages. Components "a" and "b" are implemented in Magnolisp. Component "c" has multiple target language implementations (in C#, Java, and C++), with its API declared in the `"c.rkt"` Magnolisp module. Component "d" has implementations in Magnolisp$_{C\#}$, Magnolisp$_{Java}$, and Magnolisp$_{C++}$. The Magnolisp$_{C++}$ `#lang`uage emits two submodules for `"d-cxx.rkt"`, one for its Magnolisp interface, and the other for its Magnolisp$_{C++}$ implementation. Solid boxes represent source entities, and dashed boxes represent generated ones.

Bridge languages by their nature cannot (at least not fully) abstract over their target languages, but as I argued in section 1.4.5, that loses little in portability in cases where one anyway programs natively against target APIs. Furthermore, being able to have syntactic abstractions specifically for Magnolisp-to-target bridging should facilitate target system service access, which is an important practical concern in creating useful programs.

This section's bridge languages shall also be Magnolisps, at least in the sense of stylistic uniformity and sharing of the Magnolisp$_*$ infrastructure. For most compatibility with target concepts, we could formulate our transcompiled language in terms of the target language abstract syntax, which is a strategy proven by the likes of C-Mera [Selgrad et al., 2014] and SC [Hiraishi et al., 2007]. The only major constraint for a Magnolisp's target-language likeness is that it should still have Racket's module system and adhere to Racket's scoping rules (and hygiene); that is to preserve robust composition of syntactic forms, and for compatibility with Racket's tools.

Magnolisp$_{C++}$, Magnolisp$_{Qt}$, and Magnolisp$_{Symbian}$ are less extreme in their target orientation than C-Mera and SC, in that they are strike a balance between being a Magnolisp and an S-expression-based C++ reformulation.

6.9.1 Magnolisp$_{C++}$

Magnolisp$_{C++}$ is featured in figures 6.5 and 6.6 as a language intended for implementing components that rely on C++ APIs. As a solely C++ targeting language, Magnolisp$_{C++}$ is mostly free to expose C++ features and to exploit them in its transcompilation output. That said, Magnolisp$_{C++}$ does not aim to be rich enough in C++ constructs for *creating* idiomatic C++ APIs, but rather to allow directly *using* existing C++ APIs without additional, externally-written glue code.

For more intimate C++ interfacing, Magnolisp$_{C++}$ includes a curated selection of C++ constructs, and augments Magnolisp$_r$'s C++ FFI with additional features. A particularly flexible FFI feature is an "escape hatch" allowing any function marked as `#:verbatim` to contain C++ source text as its body, to be output verbatim during transcompilation. The same kind of an FFI mechanism is supported for example by the *Ferret*[6] transcompiler, which compiles a restricted subset of Clojure to C++.

Magnolisp$_{C++}$ builds on Magnolisp$_*$ as its infrastructure: it is implemented by the `magnolisp/cxx` module, and it targets *Magnolisp$'_{cxx}$* as its core language. The dedicated core language allows Magnolisp$_{C++}$ to support C++-oriented features, which include:

- Looping expressions such as **`while`**, **`continue`**, and **`break`**.
- First-class functions, represented in terms of functor values. These are also supported by the C++ FFI, so that C++ functions can accept and return function values.
- Anonymous functions translating to C++ lambda expressions. (Magnolisp also supports **`lambda`**s, but their closures cannot escape.)
- C++ level subtyping. That is, a **`struct`**ure type defined in Magnolisp$_{C++}$ may be annotated as having one or more C++ superclasses, although

[6] `http://dropbox.nakkaya.com/builds/ferret-manual.html` (2016)

Magnolisp$_{C++}$'s type system itself does not support subtyping. This may allow calling of C++ functions requiring arguments that implement a certain C++ interface; in C++, an abstract interface may be defined as a class with only unimplemented **`virtual`** methods.

- Instance methods. Syntactically, methods are **`define`**d as functions, but with a `#:method` attribute, causing the first argument's type to name the owning class. As in STELLA, function and method calls have uniform syntax [Chalupsky and MacGregor, 1999].

For example, in Magnolisp$_{C++}$ we can use a higher-order `box-map` function in implementing a first-order function `box-add2` for Magnolisp:

```
#lang magnolisp/cxx
(provide box-add2) ; provide to magnolisp

(require ; for simulation
  (only-in racket/base box-immutable unbox +))

(declare (+ x y) #:foreign ; ad-hoc polymorphic C++ function add
  #:name add #:type (∀ E (-> E E E)))

(typedef Box-t #:foreign) ; C++ type constructor Box_t

(define (box-map b f) #:foreign ; C++ function box_map
  #:type (∀ E (-> (<> Box-t E) (-> E E) (<> Box-t E)))
  (box-immutable (f (unbox b))))

; Increments boxed value by two.
(define (box-add2 b) #:magnolisp ; supply declaration to mglc
  #:type (∀ E (-> (<> Box-t E) (<> Box-t E)))
  (box-map b (lambda (x) (+ x 2))))
```

whose `box-add2` function `mglc` would translate into C++ as

```
template <typename E>
Box_t<E> box_add2(Box_t<E> const& b) {
  return box_map(b, [] (E const& x) { return add(x, 2); });
}
```

The `box-add2` function demonstrates how Magnolisp$_{C++}$ can mediate between Magnolisp and C++ APIs. Magnolisp would be unable to directly access `box_map`, which has a Magnolisp$_{\texttt{base}}$-incompatible (higher-order function) type.

6.9.2 Magnolisp$_{Qt}$

Magnolisp$_{Qt}$ is derived from Magnolisp$_{C++}$, and provides additional syntax for the domain of Qt programming, similarly to the way that `moc` recognizes additional syntax for defining `QObject`-derived classes with **`signals`** and **`slots`**, for example. Magnolisp$_{Qt}$ automatically adds any additional C++ definitions that would normally be generated by `moc`, which simplifies the build process.

The following example function makes use of Qt's event communication mechanism by connecting a `QCompass` object's `readingChanged()` signal to an event-receiving "slot" of our own definition. Qt version 5 would allow a signal to be connected to a C++ lambda expression (or some other functor object), but a number of niche devices have shipped with Qt 4, for which only `QObject`-derived classes can implement slots. Magnolisp$_{Qt}$ works around that limitation by having a `slot-`λ construct, which compiles to an instantiation of a `QObject`-subclassing functor, for which `moc`-compatible definitions are also generated; consequently, there is no need to declare the "anonymous" functor in a header file (that is known to `moc`):

```
(define (create-compass widget)
  (define compass (make-QCompass))
  (QObject::connect compass (SIGNAL (readingChanged))
     #:qobject (slot-λ () #:parent compass
                 (update-azimuth widget
                   (*-> compass (reading) (azimuth)))))
  (*-> compass (start))
  compass)
```

whose approximate C++ translation would be

```
MGL_FUNC QCompass* create_compass(Widget* const& widget) {
  QCompass* compass = new QCompass();
  QObject::connect(compass, SIGNAL(readingChanged()),
                   new QSlotLambda_0(compass, widget),
                   SLOT(call()));
  compass->start();
  return compass;
}
```

for which a `QSlotLambda_0` functor type is additionally generated, one that implements the `call()` slot containing a translation of the `slot-`λ body.

As Qt APIs commonly deal with pointers to `QObject`-subtypes, Magnolisp$_{Qt}$ is also equipped with pointer-aware operations, such as the `*->` dereferencing call form shown above.

The compass example also demonstrates Qt's *parenting* idiom, which is commonly employed in Qt memory management (in addition to smart pointers). In that idiom the ownership of an object is passed onto another object, even when there is otherwise no containment relationship between the two; it is enough for the lifetimes of the objects to be the same. `QObject`-derived objects commonly accept an optional extra argument for the purpose of passing a pointer to an object whose ownership is to be taken; alternatively the `QObject` method `setParent` may be used. The idiom is employed in `create-compass` by marking the `compass` object as the `#:parent` of the anonymous callback object, so that both objects get deleted together.

6.9.3 Magnolisp$_{Symbian}$

Magnolisp$_{Symbian}$ is another flavor of Magnolisp that is derived from Magnolisp$_{C++}$, mostly just by having it export declarations for common Symbian data types, and additional syntax to abstract over common Symbian idioms.

One Symbian idiom is to classify certain kinds of C++ classes as *R class*es, whose instances are generally kept as automatic variables on the stack, but which also require some resource allocation to be fully usable. For this purpose, the Magnolisp$_{Symbian}$ language includes a `define+Connect/leaving` macro, which both defines a variable `x` of type `T`, and tries to do the resource allocation by invoking a `Connect` instance method, possibly with some `arguments`. The macro inserts code for *leaving* (i.e., "throwing" a Symbian exception) for reporting any failure in getting the resource:

```
(define-syntax-rule
  (define+Connect/leaving x T Connect arg ...)
  (begin
    (define x #:type T)
    (User::LeaveIfError (Connect x arg ...))))
```

An example code module written in Magnolisp$_{Symbian}$ is shown below. The module defines and exports a `set-alarm` function for Magnolisp, presumably to be used as part of a Symbian implementation of a concept for accessing system alarm services. The function uses `define+Connect/leaving` to define and initialize a client session handle to the Symbian alarm server:

```
#lang magnolisp/symbian
(require "symbian-ascli.rkt") ; RASCliSession, TASShdAlarm, etc.
(provide set-alarm) ; Symbian implementation of set-alarm

; Sets an alarm due at time, with message msg.
; Returns the ID of the set alarm, or leaves.
(define (set-alarm tim msg) #:magnolisp
  #:type (-> TTime TAlarmMessage TAlarmId)
  #:permission (WriteUserData)
  #:alert ([alarm@! #:on-leave]) ; for any error code
  (define+Connect/leaving c RASCliSession
    RASCliSession::Connect)
  (CleanupClosePushL c) ; push freeing instruction onto cleanup stack
  (define alm #:mut #:type TASShdAlarm) ; alarm details
  (set-my-alarm-Category! alm) ; product family specific category
  (TASShdAlarm::Message= alm msg) ; message string
  (TASShdAlarm::NextDueTime= alm tim) ; absolute due time
  (User::LeaveIfError (AlarmAdd c alm)) ; schedule the alarm
  (CleanupStack::PopAndDestroy) ; free c
  (TASShdAlarm::Id alm)) ; return alarm ID
```

Symbian APIs also idiomatically define *C classes*, which (possibly indirectly) derive from the class `CBase`, and whose instances are allocated from the heap, typically using an overloaded **`new`** operator that leaves on failure. An idiom related to C classes is *two-phase construction*, which involves invoking both a C++ constructor and an instance method named `ConstructL` to fully construct an object, and to do it safely so that a half-initialized object does not end up as leaked memory. It is also idiomatic to abstract over two-phase construction by implementing a **`public static`** method called `NewL`:

```
CMyClass* CMyClass::NewL(MyArg1 const& a, MyArg2 const& b) {
  CMyClass* object = new (ELeave) CMyClass(a, b);
  CleanupStack::PushL(object);
  object->ConstructL();
  CleanupStack::Pop();
  return object;
}
```

If Magnolisp$_{Symbian}$ had the core language for defining constructors and `static` methods, it might include a `define-NewL` macro for implementing both a C++ constructor and a `NewL` method with matching argument lists. However, as Magnolisp$_{Symbian}$ merely aims to support *using* (not implementing) idiomatic Symbian C++ APIs, the language merely includes `two-phase-make/leaving` expression syntax for performing two-phase construction at the API call site, thus letting it get away with less core language:

```
(define-syntax-rule
  (two-phase-make/leaving make-T/leaving ConstructL arg ...)
  (let ([object (make-T/leaving arg ...)])
    (CleanupStack::PushL object)
    (*-> ConstructL object)
    (CleanupStack::Pop object)
    object))
```

Any identifier passed in as the `make-T/leaving` macro argument can simply be bound to a declaration of a `#:foreign` "function," whose name translates into C++ as `new (ELeave)` *T*, where *T* names the instantiated type. Thus, Magnolisp$_{Symbian}$'s C++ FFI does not necessarily need to support C++ operators or their overloading.

While the above syntax definitions were basic enough to be implementable without any Symbian-specific core language constructs, the processing required by Magnolisp$_{Symbian}$ is nonetheless not entirely free of Symbian specifics. In particular, when analyzing its core language, it is necessary for `mglc` to account for the possibility of non-local returns occurring due to leaves.

6.10 Macro-Based Mapped Types

Our Magnolisp flavors are similar enough that they may be able to accommodate common macro-implemented facilities. The component system is one such facility that might find use beyond Magnolisp$_r$ for managing code, while mapped types is another macro-implementable and reusable mechanism of the kind that is often useful in a cross-platform setting.

Mapped types, as found in the commercial RemObjects Elements language environment, facilitate the generation of source code optimized for readability by someone familiar with the target language; specifically, a *mapped type* may be used to map a source language type name and uses of its operations into the appropriate target language identifiers and expressions, so that a source language API is effectively implemented as a mapping rather than as a library.

Where desirable, we can define such mappings for a Magnolisp$_{lang}$ in terms of basic pattern macros, or perhaps in terms of a `define-mapped-type`

convenience macro, which takes either identifier-to-identifier or expression-to-expression transformation rules. Suppose, for instance, that we have declared two string APIs: the C++ standard `std::wstring`, and the Qt-specific `QString`. We might then define a mapped `String` type, which uses the more fitting choice for our target:

```
(static-cond
 [qt?
  (define-mapped-type String #:mapped-to QString
    [make-string #:mapped-to make-QString]
    [string-index #:mapped-to QString-indexOf]
    [string-prefix? #:mapped-to QString-startsWith])]
 [cxx?
  (define-mapped-type String #:mapped-to std::wstring
    [make-string #:mapped-to make-std::wstring]
    [string-index #:mapped-to std::wstring-find]
    [(string-prefix? s pfx) ; no direct equivalent operation
     #:mapped-to
     ; instead expand to s.compare(0, pfx.size(), pfx) == 0
     (let ((x pfx))
       (= (std::wstring-compare s 0 (std::wstring-size x) x)
          0))])])
```

With such definitions, and any referenced `#:foreign` declarations, the code

```
(define (f s x y)
  #:type (-> String String String String)
  (if (and (>= (string-find s x) 0)
           (string-starts-with? s y))
      s "no"))
```

would translate for a non-Qt C++ target roughly as

```
std::wstring f( std::wstring const& s,
                std::wstring const& x,
                std::wstring const& y ) {
  return ((s.find(x) >= 0) ?
          (s.compare(0, y.size(), y) == 0) :
          false) ?
      s : std::wstring(L"no");
}
```

6.11 Resolving Build Dependencies

Already Magnolisp^{v0} includes functionality for inferring build dependency information based on used types and functions and their `build` annotations. Magnolisp$_*$ likewise supports such inference for Magnolisp$_{base}$ languages, and augments that support by also collating any annotated `#:permission` requirements, similarly to the solution described for Magnolia in chapter 4.

Upon request, `mglc` resolves the build dependencies for a program, and outputs them as include files in the requested supported languages; this process is illustrated in figure 6.7. Even when programming against abstract

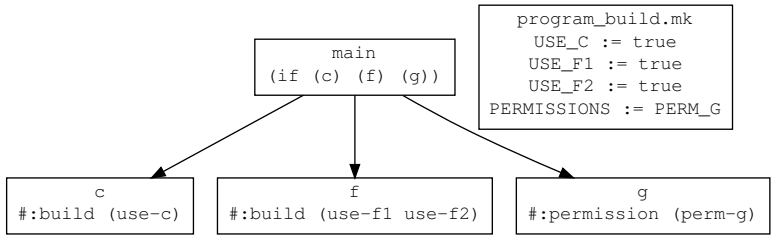


Figure 6.7: A program's `main` function that calls `#:foreign` functions with build and permission requirements. The resolution results can be output by `mglc` in the GNU Make language, among others.

interfaces (of concepts), specific implementations will be resolved at compile time, allowing `mglc` to infer build (and permission) requirements.

Magnolisp's `#:build` requirements are annotated per type and function, which results in relatively fine granularity in deducing the requirements; if a component implements multiple operations, and only some of those operations require a certain library, then a program may use the other operations of the component without introducing a dependency to the library. The unused definitions are removed during whole-program optimization, and only the remaining ones contribute to the inferred dependency information.

As suggested in section 1.3.2, `#:build` requirements may be annotated in terms of abstract names, each indicating a dependency; this makes it less work to repeat an annotation for multiple definitions of the same API, and it also makes it possible to specify different actual build instructions for each target. As an example, we might specify a `use-sqlite` dependency for a type, and as a consequence, using any function involving values of that type will introduce the dependency for a program:

```
(typedef Sqlite ; SQLite database connection handle
  #:foreign #:name sqlite3 #:build (use-sqlite))
```

The required `#:permissions` for a function may be specified as a list (expressing a set), or as an expression which may use `#:and` and `#:or` as logical connectives. The Magnolisp compiler collates the individual specifications into a total set of permissions to request for a program, except that it is up to build scripts to compute desired permission requests when `#:or` expressions are involved. The exact same implementation of a function might run on different targets and require different permissions; this can be handled though conditional compilation, for example with our `static-cond` construct:

```
(define (start-bluetooth-discovery bt)
  #:foreign #:type (-> BluetoothAdapter Bool)
  #:alert ([bluetooth@! #:post-unless value]))
(static-cond
  [(<= api-level 8) ; Android 2.2 (API level 8)
   (declare start-bluetooth-discovery
```

```
    #:permission (BLUETOOTH))]
  [else ; Android 2.3 (API level 9)
   (declare start-bluetooth-discovery
    #:permission (BLUETOOTH BLUETOOTH_ADMIN))])
```

If permission requests cannot be assumed granted, then run-time errors due to lack of permissions are possible. Declaring alerts for error reporting is orthogonal to declaring permissions. In the case of `start-bluetooth-discovery`, the primitive is assumed to report any error as a `#f` value, which translates as a `'bluetooth@!` alert. Using the `@!` suffix in naming alerts is a Magnolisp^{v1} convention.

6.12 Capturing Build Domain Knowledge

After `mglc` has resolved a set of build dependencies, any symbolic ones (such as `use-sqlite`) must be translated to concrete ones (e.g., source files and flags for compilation, library names for linking, etc.) for building. In this chapter's architecture, that translation is left for a product-line-specific makefile to determine, probably depending on current configuration parameters for their information about the target platform and build toolchain.

The widely used *GNU Make*[7] build manager, for instance, has sufficient capabilities for driving the build process, and for capturing platform-specific build knowledge. Additionally, if some of the necessary knowledge is already captured by existing domain-specific tools, one may want to have the product line's top-level makefile invoke those tools, in order to either extract the relevant available information, or to run some of the build under an external tool.

For building a Magnolisp program for a target, one would add a Make rule for invoking `mglc` on the source program, so that it generates both a target language program and build dependency information. A nested `make` invocation can then **`include`** the dependency information, and proceed to build the dependencies, and then build and link the program binary. Any further packaging of the program can then be done, perhaps with vendor-provided tools, and this process may also require some of the dependency information, such as any requirements for requesting permissions (or hardware capabilities).

One aspect of dealing with concrete build dependencies is making sure to import the necessary APIs for the generated target language source files. In Magnolisp, the way this is done varies depending on the target; some targets require additional `#:import` annotations for Magnolisp primitives, to allow any resolved imports to already be emitted by `mglc` during transcompilation (e.g., using **`import`** in Java).

For C++ targets, the programmer may freely decide how to **`#include`** files or to forward declare names for a particular symbolic `build` dependency, and this can be done differently for interfaces and implementations, as there is a `MGL_IMPLEMENTATION` macro only **`#define`**d for the latter. For example, if compiling `"engine.rkt"` for a C++ target, `mglc` would generate a `"engine.hpp"` file that **`#include`**s `"engine_config.hpp"`; in that file, then, we can include any necessary header for `use-sqlite` by writing

[7] `https://www.gnu.org/software/make/`

```
#include "engine_build.h" // import dependency information

#ifdef USE_SQLITE
#include <sqlite3.h>
#endif
```

Another aspect of abstract-to-concrete build dependency translation is to decide how to build a dependency and link it into the program. We might have to specify conditional addition of **#include** paths for our program, or conditional building and linking of targets such as object files or static or dynamic libraries. For example, we might satisfy the `use-sqlite` dependency in terms of an existing dynamic library; in a GNU Make file we might specify the appropriate linker option as

```
LDFLAGS += $(and $(USE_SQLITE), -lsqlite3)
```

In a `qmake` file, in turn, we might specify that option as

```
USE_SQLITE {
  LIBS += -lsqlite3
}
```

Where using a subordinate build manager lacking conditional compilation, we can use a separate preprocessor to enable conditionality. For example, we might use a Ruby-based template engine to allow for conditions in a Symbian MMP file:

```
<% if $USE_SQLITE %>
library sqlite3.lib
<% end %>
```

The popular `pkg-config`[8] tool can be useful for querying information about locally installed libraries on some systems; its data is maintained in a modular form, as one metadata file (named "*name*`.pc`") per package. For example, to query the appropriate compile and link options for the GNOME object system API, we might write (in GNU Make)

```
ifeq ($(USE_GOBJECT),true)
  CFLAGS += `pkg-config --cflags gobject-2.0`
  LDFLAGS += `pkg-config --libs gobject-2.0`
endif
```

For maintaining specific build information we end up using multiple different languages and tools. It might not be that useful to try to consolidate on a uniform representation for that information, since it—by its nature—is target specific, as is the availability of any tools used to extract or use the necessary information. The drawback of dealing with bespoke build description languages is that Magnolisp editing environments such as DrRacket cannot out of the box offer any language-aware editing support.

[8] `http://pkg-config.freedesktop.org/`

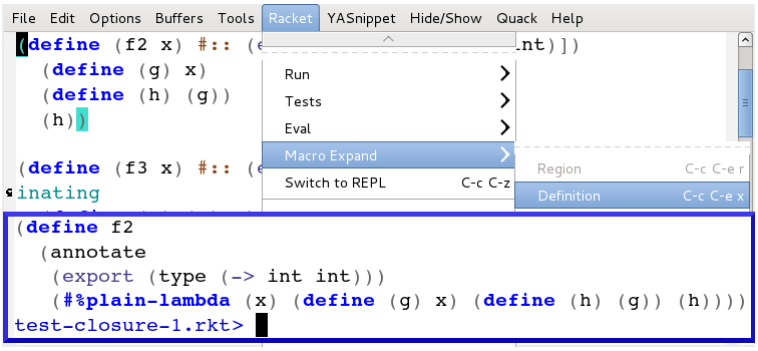


Figure 6.8: Macroexpanding a **define** form in Magnolisp, using Emacs with racket-mode.

6.13 A Product-Line Development Environment

Basing our product-line programming language infrastructure on Magnolisp* helps consolidate most programming into a single language environment, which can largely be assembled out of existing Racket tooling.

As Magnolisp*-based languages are also Racket-based languages, existing Racket tools and editors can support them as such; any language-specific support naturally requires customization. Scribble [Flatt et al., 2009], for example, is readily usable as a tool for documenting Magnolisp$_{lang}$ APIs, although it could benefit from additional syntax for documenting Magnolisp concepts and implementations; that syntax can be defined as macros. The code navigation and refactoring support found in the DrRacket IDE [Findler et al., 2002] and Emacs' racket-mode is directly usable with Magnolisp and its variants.

For example, racket-mode has a command for incrementally expanding macro uses, as shown in figure 6.8, and that functionality works for Magnolisp code just as it does for Racket code, since the macro system is the same. Similarly, racket-mode's **require** refactoring can also transform Magnolisp code, since Magnolisp's module system is the same as Racket's.

Section 6.4's component system might benefit from hover documentation displayed by an editor, just as it is useful for Magnolia's Eclipse plugin [Bagge, 2013] to display Magnolia component details. Magnolisp's macro-based component implementation means that information about components disappears during expansion, unless it is specifically preserved. In the same manner as Magnolisp languages' macros record information for purposes of transcompilation, information about components could be stored into a submodule, so that it can loaded on demand by an IDE, and rendered for display to the user.

CHAPTER 7

Discussion

This dissertation builds on a line of work on the design, implementation, and use of mouldable programming languages. That work so far has experimented largely with Magnolia and C++ as languages that either manifest or can accommodate language in support of the mouldable programming methodology. This work has complemented those two languages with Magnolisp, which brings the flexibility and adaptability aspects of mouldability into its specific focus, also with respect to its own implementation. The genericity [Bagge and Haveraaen, 2010, 2014] and robustness [Bagge and Haveraaen, 2009; Bagge et al., 2006, 2009] aspects of mouldability have received more attention in prior work.

My niche platform software product-line strategy revolves around a mouldable programming language as a key production tool. I have no proper evaluation of the strategy to offer, but in this chapter I discuss its known uncertainties and perceived benefits and shortcomings. I complement the discussion by reviewing a selection of related work, from which the interested reader may seek ideas for technological and methodological improvements to the solutions presented in this dissertation.

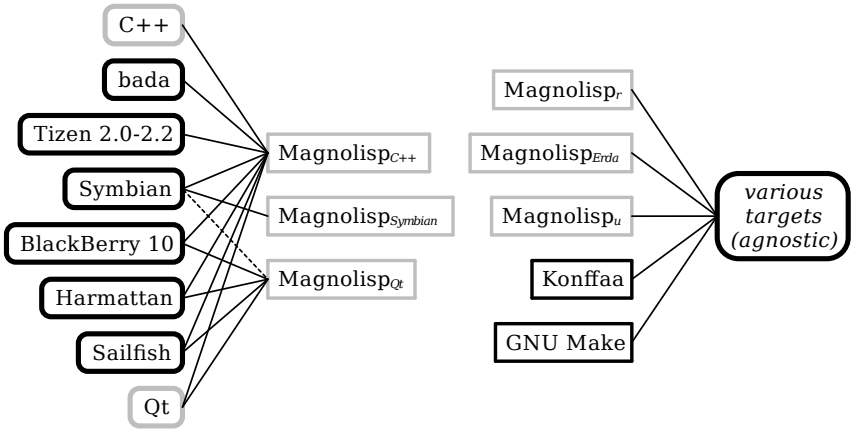
7.1 Benefits, Shortcomings, and Uncertainties

> "Unfortunately, no one can be told what the Racket is. You have to see it for yourself."
>
> Andrew M. Kent, paraphrasing Morpheus in The Matrix

Magnolisp's bottom-up implementation in terms of Racket's language definition facilities makes it not only a language, or a family of related languages, but also infrastructure for implementing more members of that family. Magnolisp follows in the Lisp tradition of pragmatic, lightweight, reusable language solutions, while still attempting to be uncompromising when it comes to properties that might help in scaling to large or complex product lines: it shuns product-line-encompassing "projects" to allow focusing on one program family member at a time; it supports separate source-to-IR compilation of core

assets, which might be many; and it features lexically scoped macros for safer use of multiple syntactic abstractions simultaneously.

Chapter 6 sketched a design for a family of Magnolisp-based programming languages, and their use as part of a product line, together with auxiliary tools (such as configuration and build managers). The level of detail in that design, and the fact that some of its building blocks have already been created, inspire some confidence in my PLA strategy's suitability for addressing various cross-niche-platform product management needs.



It should also be possible to discover suitable language designs for chapter 6's language family, as we can learn from existing languages of a comparable nature: the restricted and target-agnostic Magnolisp$_r$ language is similar to Magnolia; the richer and still target-agnostic Magnolisp$_u$ is not unlike the multi-targetable STELLA language; and the target-specific Magnolisp$_{C++}$ and Magnolisp$_{Qt}$ languages are similar in nature to C-Mera, for example.

There is some uncertainty regarding the extent to which we can adopt existing solutions in implementing Magnolisps. First, the source languages feature hygienic macros (unlike, e.g., Magnolia, STELLA, or C-Mera), and this may necessitate or encourage different design and implementation choices. Second, the niche-platform PLE context calls for extensive FFI facilities, since the platform landscape is heterogeneous, with multiple target languages, each of which we may want to target natively. Furthermore, our component system calls for code parameterization support, and the FFI should not be inconvenient in the target language either; I expect design challenges in this area, at least for some target languages.

Given the heterogeneous platform setting, I consider it important to enable syntactic abstraction over various platform conventions. In section 6.9.2, I sketched some syntax for more convenient and portable use of Qt's slots and signals mechanism, for example, and I previously proposed [Hasu, 2012] similarly motivated language-based abstraction over specific error handling mechanisms; both of these example use cases are likely to require highly target-specific core language. Having a range of languages ranging from platform agnostic to specific, as described in chapter 6, should arrange for compartmentalized access to various target language mechanisms. That approach should hopefully allow for the definition of a variety of syntax for capturing platform idioms, while still helping to contain the richness of individual core languages.

7.1.1 "Multi-Core" Transcompiler Engineering

I have not presented a systematic approach for maintaining transcompilation routines for multiple different core languages within a single compilation environment. Solutions for modular and composable compilation do exist (e.g., Silver), but it is not certain that they would enable much reuse in this case, compared to less advanced implementation techniques. Chapter 6's PLA calls for either highly similar or significantly different languages: the similar ones (e.g., Magnolisp$_{C++}$ vs. Magnolisp$_{Qt}$) might even share the same core language, or at least the same compilation pipeline (in a different mode); whereas the different ones (e.g., Magnolisp$_r$ vs. Magnolisp$_u$) involve different analysis and optimization possibilities, and might have very limited commonalities in their sequences of compilation steps, except perhaps towards the back end for common target languages.

Our core languages would ideally be of manageable size, so that even maintaining their compilation routines entirely separately would be feasible. Even if so, we would probably want to enable at least ad-hoc code reuse between core language processors; this might for instance be done by organizing our compilation code into small routines of limited concerns that are more likely to be reusable in different contexts, or organizing it into generic or configurable routines capable of adapting to different contexts.

Concrete examples of potential code-organization tactics to employ have also been presented in this dissertation's chapter 3 and in our paper on code formatting [Bagge and Hasu, 2013], although without reports of their extensive practical application, which means that those tactics remain mostly unproven for now.

Chapter 3 discussed achieving ad-hoc genericity by declaratively implementing special casing for multiple AST node types, in an open-ended way. There is little doubt that this results in more abstraction and less special casing in the `mglc` codebase, thus making routines both shorter and more general. However, the codebase presently fails to make extensive use of the more unorthodox capabilities of the Illusyn library, to allow for better assessment of the usefulness of those capabilities; one reason for this is that I (as a compiler engineer) am not used to having those capabilities available, and have yet to develop practices for exploiting them.

In our code formatting paper [Bagge and Hasu, 2013] we discussed achieving concern separation by organizing code as pipelines of token processors. The Magnolisp compiler's internal use of token streams in code formatting is presently limited to an adaptation of the algorithm described by Kiselyov et al. [2012], which tackles the notorious pretty-printing problem through incremental stream processing.

The current version of `mglc` has few back ends, and little pressing need for code reuse in code formatting. My PLA strategy calls for multiple back ends, however, and the pipelines-of-token-processors approach might be particularly attractive if one wished to quickly implement basic code formatting for multiple similarly styled languages; one might achieve that simply by having different back ends emit correctly categorized tokens, and feeding them into a common, suitably configured pipeline. One fortunate state of affairs is that at least with respect to code formatting, the current mobile platforms constitute a relatively uniform target, as almost all of their vendor-supported

programming languages have a C-style concrete syntax, with similarities in their spacing and line breaking rules.

7.2 Related Work

The Lisp tradition of pragmatism that has inspired Magnolisp is clearly manifested in the small implementation of *C-Mera* [Selgrad et al., 2014] (previously known as *CGen*) and the powerful applications it nonetheless allows [Lier et al., 2016; Selgrad et al., 2016]. C-Mera is a Common Lisp embedded transcompiled language. It is essentially just C with S-expression syntax (and support for self extension with Common Lisp macros), which means that no sophisticated analyses are required for compiling it to C. That characteristic also makes it straightforward to add AST node types to support additional languages whose syntax is similar to that of C, with C-Mera already featuring modules for C++, CUDA, GLSL, and OpenCL. The success of C-Mera's implementation strategy suggests that a single, straightforward compilation pipeline might be adequate for supporting multiple related target-language-specialized languages such as Magnolisp$_{C++}$ and Magnolisp$_{Qt}$ of chapter 6.

LambdaNative [Petersen et al., 2013] is a cross-platform application framework implemented in Scheme. It is able to target multiple mobile OSes, and has been used to develop a number of applications for the mHealth domain in particular. LambdaNative relies on Gambit-C for transcompiling Scheme programs to C for further compilation with a target-compatible C compiler. The portability of the C code depends on the availability of standard UNIX and POSIX APIs, and the APIs of the framework's own portable libraries such as a GUI widget library targeting OpenGL (ES). In contrast, my product-line strategy is aimed at targeting native languages and frameworks, for maximum compatibility, but with various forms of abstraction to help manage target-specific details.

RemObjects Elements (version 8.3), as already discussed in section 1.4.5, has a cross-platform strategy similar to mine, as it has specific support for targeting platforms "natively;" it does not support C or C++ as targets, however, meaning that it is not especially suitable for natively targeting niche smartphone platforms such as BB10, Sailfish, and Tizen. All of the Elements-based languages share the same language environment (either Visual Studio or the dedicated Fire IDE), which is like my strategy of designing Magnolisp variants for compatibility with Racket tooling; however, my strategy is to have a language family with domain-oriented variation, while the Elements languages' differences are mostly stylistic. Elements' source (and target) languages are all object oriented, with their facilities for API-level abstraction and code composition being of a different nature than those of Magnolia, which are based on ADTs and components. Elements languages lack macros, but they may use "aspects," which are a form of compiler extension; an aspect's implementation is loaded at compile time to influence the generation of code for class or member definitions annotated with that aspect.

PureScript[1] is a Haskell-resembling programming language that compiles to JavaScript. The language is designed for fairly direct, human-readable

[1] `http://www.purescript.org/` (2016)

translation, and the design appears quite target-language agnostic, as suggested by various efforts to create alternative back ends for it. PureScript appears to have a convenient JavaScript FFI, as suggested by the large number of available bindings to JavaScript libraries. PureScript facilitates static reasoning in terms of its advanced static type system. Magnolisp likewise aims for readability of generated code, target-language agnosticity, having lightweight FFI mechanisms, and ease of static reasoning, although its support for static reasoning is not based on type system sophistication.

nesC [Gay et al., 2003] is an example of a domain-oriented language; it is for the domain of networked embedded systems, and has been used to implement the *TinyOS* operating system for sensor networks. Like Magnolia, nesC supports component-oriented application design. It is also similar to Magnolia in that its programming model is suitably restricted to enable a larger class of accurate static analyses; the nesC compiler is capable of detecting most data races at compile time, for example, and performs whole-program inlining and dead-code elimination, before transcompiling to C. Despite its restrictions, nesC is expressive enough for writing real-world applications in its resource-constrained domain; my strategy is to allow the programmer to make a tradeoff between accurate static analysis and expressive power, through a choice of languages of different characteristics.

Silver [Wyk et al., 2010] is an attribute grammar specification language, which may be used to specify languages in a modular manner. As attribute grammar rules may also derive a translation to a target language, Silver is capable of transcompilation, and has been used to implement an extensible-C-to-C compiler, for example [Williams et al., 2014]. An advantage of Silver over Racket is that it supports the definition of new (core) language constructs, including their semantics, with any associated static analyses or optimizations; this makes it candidate infrastructure for maintaining a family of cores for related languages. Silver also supports "forwarding" [Van Wyk et al., 2002], which allows language constructs' semantics to be defined by translation to other constructs, similarly to macros; even where forwarding is present, some semantics may still be specified explicitly as attributes, which might be a useful possibility for extensions to reasoning-focused languages like Magnolia.

Magnolia In chapter 6 I envisioned a PLA with Magnolisp as its central production tool; had I instead picked Magnolia for that role, that would have called for solutions for managing multiple different variants of that language.

Bagge has previously presented the interesting idea of creating a domain-specific language specifically for the purpose of maintaining variants of a particular language (family). Her *MetaMagnolia* [Bagge, 2010a] domain-specific language (DSL) functions as a description language for specifying the concrete syntax and static semantics for variants of Magnolia. MetaMagnolia cannot be used to define new core syntax; the dynamic semantics of a MetaMagnolia-defined language is determined by translation to existing "Magnolia Core" language (or parts thereof). A MetaMagnolia-based language specification is complete enough to allow corresponding language front-end implementation code to be generated based on it, and the MetaMagnolia compiler does that by generating code in the SDF and Stratego languages. MetaMagnolia has no support for the definition of self-extensible Magnolias.

Bagge [2010b] has previously also proposed a language self-extension scheme specifically for Magnolia. It supports both "operation patterns" and "transforms," which roughly correspond to pattern macros and programmable macros in Racket, with the exception that Bagge's transforms support multiple extension points and traversal modes. Racket's macro-expansion time corresponds only to the **`desugar`**ing extension point, with expansion defaulting to **`outermost`** order; other traversal modes are possible due to programmability, but bottom-up traversals are incompatible with macro hygiene. Bagge's scheme has no specific support for maintaining hygiene.

Sugar* *Sugar** [Erdweg and Rieger, 2013] is a framework for turning non-extensible languages into extensible ones. The resulting languages are extensible from within themselves, in a modular way, so that extensions are in scope following their respective module imports. Sugar* extensions are not mere macros (i.e., functions which translate the extended syntax away during parsing), but rather they extend the base language grammar with new productions, and define associated AST nodes and desugarings; in addition to syntax, an extension may also introduce static analyses and editor services. Any safety of extension composition relies on user-imposed discipline in defining them. Sugar* languages' editor services are available within Eclipse, and realized through integration with the Spoofax [Kats and Visser, 2010] language workbench.

Sugar* meets most of the language engineering needs of my product-line strategy, with some advantages (e.g., mechanisms for defining semantics for new "core" language) and disadvantages (e.g., lack of extension composition safety) compared to Racket. Due to its ability to extend existing languages, Sugar* might also enable the use of extensible target-platform-*native* languages within a product line; an extensible Java, for example, is already available in the form of *SugarJ* [Erdweg et al., 2011].

mbeddr *mbeddr* [Voelter et al., 2013a] is a feature-rich programming language and IDE for the domain of embedded systems programming; its technology stack builds on the JetBrains MPS language workbench. mbeddr's programming language is *mbeddr C*, which resembles C closely [Voelter et al., 2012], and transcompiles to C for deployment to embedded targets. Focusing solely on C as a target language for programs gives good reach over the embedded domain, as C has long been the dominant embedded programming language [UBM Electronics, 2012].

The niche smartphone situation is different in that target platforms tend to ship with a preinstalled, locked-down OS, with a de-facto-favored language. My chosen strategy differs from mbeddr's by using target-language-agnostic source languages for composing systems and programming application logic, so that each platform can be targeted via its native language. My strategy also entails using multiple target-specific languages, rather than just one specialized for C. A commonality between the two approaches is the idea of domain-specific language extension, although my approach also supports self-extension, perhaps leading to more pervasive and context-sensitive syntax extension.

The mbeddr C language is not extensible from within itself, not even in terms of the C preprocessor, which it does not include. However, mbeddr C is extensible externally (in terms of JetBrains MPS) in a modular and principled way, so that each extension can define its syntax (both notation and abstract syntax), type checking, dynamic semantics, and IDE support. The IDE features "projectional editing" with various alternative notational styles, including text, prose, math tables and graphics [Voelter and Lisson, 2014]; such choice of notations is unavailable to Magnolisp, but various alternative textual notations can easily be defined in terms of macros, even for local use, and perhaps even specializing for local names.

While reasoning about Magnolia is based on generic language restrictions and declared API semantics, mbeddr authors advocate static analysis of DSL code, as domain-specific constructs would tend to more restricted and richer in high-level semantics than general-purpose expression language. mbeddr features language for state machines, for example, and such language can be complemented with additional language extensions related to describing verification conditions, to support generation of input for external C verification tools [Molotnikov et al., 2014]. An extensible Magnolia's static analysis capabilities could be further improved using the same approach. Another potential application of the general idea would be to give a language such as Magnolisp more powerful core syntax, while enforcing restricted use patterns in terms of domain-specific abstractions defined as macros (restricted, e.g., to maintain analyzability or compiler or tool expectations).

mbeddr has been designed to function as a comprehensive product-line engineering environment [Voelter and Visser, 2011], as suggested by its specific support for requirements and product line variability management, for instance. mbeddr includes a DSL for describing requirements, and implementation artifacts may point to associated requirements to support "tracing," for example [Voelter et al., 2013b]; including similar annotations in Magnolia or Magnolisp would also be possible. mbeddr also includes language for expressing product configurations; unlike with Konffaa, sets of all possible valid configurations are specified separately (as **`feature model`**s) from individual configuration instantiations [Tomassetti and Ratiu, 2013].

mbeddr's approach to developing domain-specific software engineering tools is "Generic Tools, Specific Languages" [Voelter, 2014], which is founded on the idea of adapting languages for purposes of domain and tool integration, rather than adapting engineering tools to the needs of a domain. My strategy also subscribes to that general idea, but it does not specifically insist on a language workbench as its technological basis, nor on an IDE that is aware of the domain-specific languages and their semantics.

Haxe *Haxe*[2] is a prominent example of a general-purpose programming language transcompiling to multiple targets. Haxe has back ends for C++, C#, Java, Lua, Python, etc., and this extensive selection of targets makes it suitable for targeting various niche smartphone platforms in their native languages, which is also my strategy.

As support for implementing cross-target abstractions, Haxe features a dedicated conditional compilation mechanism, in the form of the **`#if`** .... **`#el-`**

[2] `http://haxe.org/` (2016)

`seif` .... **`#else`** .... **`#end`** construct, with access to compiler flags from associated conditional expressions. Haxe also supports "abstract types," which are similar to mapped types in that they are a compile-time feature for defining types over concrete types (either directly or via other **`abstract`** types). Some Haxe constructs can be annotated with "metadata," and there is distinct annotation syntax for run- and compile-time inspectable metadata; compile-time metadata can communicate information to the Haxe compiler, or to macros. Haxe supports various kinds of **`macro`**s as a self-extension facility, and has a rich `haxe.macro` API for compile-time AST manipulation.

The Haxe language is similar in nature to chapter 6's Magnolisp$_u$, as due to its expressivity and portability, it may be used to write portable application logic for multiple targets. Haxe's standard library includes a selection of cross-platform APIs for data structures and algorithms, which are available for all targets. The library also has some cross-platform APIs for accessing system services, available for a subset of Haxe's targets. A third category of APIs in the library is target-specific APIs, implemented only for a single target, and typically concerning highly target-specific functionality (e.g., web browser, Flash Player, or jQuery library integration for the JavaScript target). Additional Haxe APIs such as OpenFL, NME, StablexUI, and HaxeUI are available, and might be used for portable implementation of user interfaces for smartphone targets, as long as it is not a requirement to use target-native widgets.

CHAPTER 8

Conclusion

I am a proponent of choice, but choice can be expensive to enable; seldom does an end user have much choice between operating systems on which to run a specific software application, for instance. The work on this dissertation has been motivated by the challenges of cross-platform software development targeting niche platforms. I have tackled those challenges in the hopes of making it less costly to provide choice.

Getting intimately familiar with vendor-specific application developer offerings is a risky investment when it comes to a platform that could be announced discontinued at any time. Even worse, such offerings can differ greatly between platforms, making for an inconsistent cross-platform development experience, often with few obvious code reuse opportunities. Existing cross-platform software development solutions, on the other hand, typically only include support for more mainstream platforms.

My belief is that these challenges cannot be entirely overcome, but that programming language technology can make targeting an additional platform a more systematic, predictable, and contained experience, at least for those who are prepared to engage in some language engineering. By "contained" I refer to limiting the extent to which one must acquire and remember platform-specific knowledge.

My assumption is that product-line-oriented interfaces and languages can be used to abstract over vendor-supported interfaces and languages, meaning that while the required target details must be looked up for purposes of implementation, that information can be captured inside interface and language implementations (of product-line-conventional naming and syntax), to avoid the need to constantly recall it. The product-line-oriented languages can have their own development environment, meaning that there is less need to become intimately familiar with the nuances and quirks of multiple vendor-provided environments. The source languages can be implemented in terms of translation to target languages, thus allowing the use of vendor-provided build tools, and avoiding the need to learn about target binary formats.

In this dissertation we have described a number of solutions that I imagine could function as building blocks for pragmatic implementations of such product-line-oriented languages and integrated tooling. Those solutions seek

to facilitate many aspects of the implementation of cross-platform-deployment and product-line friendly languages, including surface syntax definition and extension, program analysis and transformation, third-party language infrastructure reuse, and portable error handling.

The presented solutions do not constitute a comprehensive set of building blocks for the product-line language vision, however. I did not cover concurrency, for example, which is a central concern for many kinds of reactive (or user-responsive) applications. Neither did I cover parallelism, which is becoming an increasingly important concern for application developers, given the nature of modern hardware architectures. For convenient development of multi-tasking applications, we might wish for cross-platform-deployment-friendly language support for concurrency and parallelism; both can be made more manageable through language design, as suggested by the favorable characteristics of languages such as Elm [Czaplicki and Chong, 2013] and ParaSail[1].

While I have made some educated[2] guesses regarding the usefulness of the presented solutions in the niche platform setting, I cannot be certain that they would actually come together to achieve the product language family vision outlined in this dissertation, nor can I be sure that the complete solution would be especially fit for purpose. Acquiring such certainty is hard to do without actually creating a software product line of the envisioned kind, but product creation is beyond the scope of mere PhD work; I shall leave that for future work in a more industrial setting. For now, I trust that the presented technologies stand on their own merits, and may even happen to have applications far outside the niche platform domain.

8.1 Contributions

I have presented various technologies with applications to creating language-based tools for niche platform software development. The main contributions of this dissertation are:

- A technique for exploiting fairly Racket-specific macro features (subform expansion and submodules in particular) to allow a macro system to be reused as another language's syntactic extension mechanism, in a manner that retains the possibility of separate (per-module) compilation. We showed that macros can take advantage of the module-compilation time to prepare for further compilation to a third language.
- A scheme for lightweight, declarative, language-integrated AST API generation, such that it is possible to generate not only concrete (data storing) node types, but also abstract ones (abstracting over related concrete types) of a similar appearance, complete with interfaces and "representations."
- A language-based solution for API-access permission management, relying on a static-reasoning-friendly programming language to allow for complete permission requirement inference, and on a static component system in order to support cross-platform codebases.

[1] `http://www.parasail-lang.org/`

[2] My "education" in these matters comes from personal experience in developing software for niche platforms such as GEOS, SIBO, Symbian OS, Maemo, and MeeGo Harmattan.

- A design for a general, platform-agnostic failure management convention that is static reasoning friendly, and programming language support for syntactic convenience in following the convention and adapting to it.

8.2 Niche Platform Strategy Summary

This dissertation has presented various programming-language-focused technologies developed with cross-niche-platform application authoring and maintenance in mind. According to this dissertation's software production strategy, those technologies—as summarized in section 8.1—have several application areas within the niche-platform software development domain. In particular, they can be applied to constructing language-based tools that help treat different platforms the same, or—where that is not feasible—help deal with platform specifics.

Portability and Platform Abstraction: To avoid having to write application logic separately for each niche platform, it should be written portably; for that, abstraction is essential, and language is a tool for that:

- Language can abstract over target languages, through source-to-source compilation.
- Language can abstract over the use of specific implementations, through a distinction between interfaces and implementations, as can be made by a component system.
- Language can abstract over idiomatic design patterns, by allowing syntactic abstractions to be defined in the form of macros, which can then emit syntax to implement specific patterns.
- Language can abstract over language implementation details, to make their application logic more reusable; for example, there can be language for defining intermediate language representations, whose data may be exposed in an abstract and controlled fashion though multiple alternative interfaces (and views).

Product Family Engineering: Software assets for a product family may be encapsulated as components, and composition of whole products need not entail run-time overhead if the component system allows it to be done statically, at compile time. To allow assets to compose, they must be compatible at the interface level, and it helps to have common conventions in particular for universal concerns affecting most components; specific support for such conventions can be included in a programming language, even for cross-cutting concerns such error handling. Language constructs can also be turned into modular assets, possible to organize into libraries, if they can be expressed as syntactic (and preferably hygienic) macros; such constructs will work across target languages if they get translated via a common core language.

Configuration Management: In a language with a component system, product configurations can in large part be expressed as programs composed out of compatible software assets. Axioms declared for component interfaces can

help statically verify or dynamically test the validity of program compositions. When component implementations are annotated with the relevant attributes, and the composition language permits resolution of (potentially) invoked operations statically, then it is possible for whole-program analysis to infer configuration information such as the set of required permissions; that information then need not be manually declared and separately maintained for each product variant for purposes of building (or publishing).

8.3 Software API Summary

This dissertation has discussed various pieces of proof-of-concept software, including Illusyn, Magnolisp, Erda$_{GA}$, Erda$_{C++}$, and Konffaa. The following diagrams summarize those software pieces' central application-specific APIs and languages, by listing their "vocabulary" (i.e., exported symbols). I hope that this gives some idea of the size, scope, and functionality of those interfaces.

In listing exported symbols, the diagrams show transformer and value binding exports in a different style, with the former kind typically being macros. Symbols that are intended primarily for use by macro implementations have the number "1" as a superscript. Arrows go from API clients to API providers. APIs without labels are for implementers rather than users. The `mglc` and `konffaa` tools expose a command-line interface rather than a programming interface, and thus no exported symbols are listed for them.

Further documentation, source code, and installable software packages for Illusyn, Magnolisp, the Erda languages, and Konffaa are available from

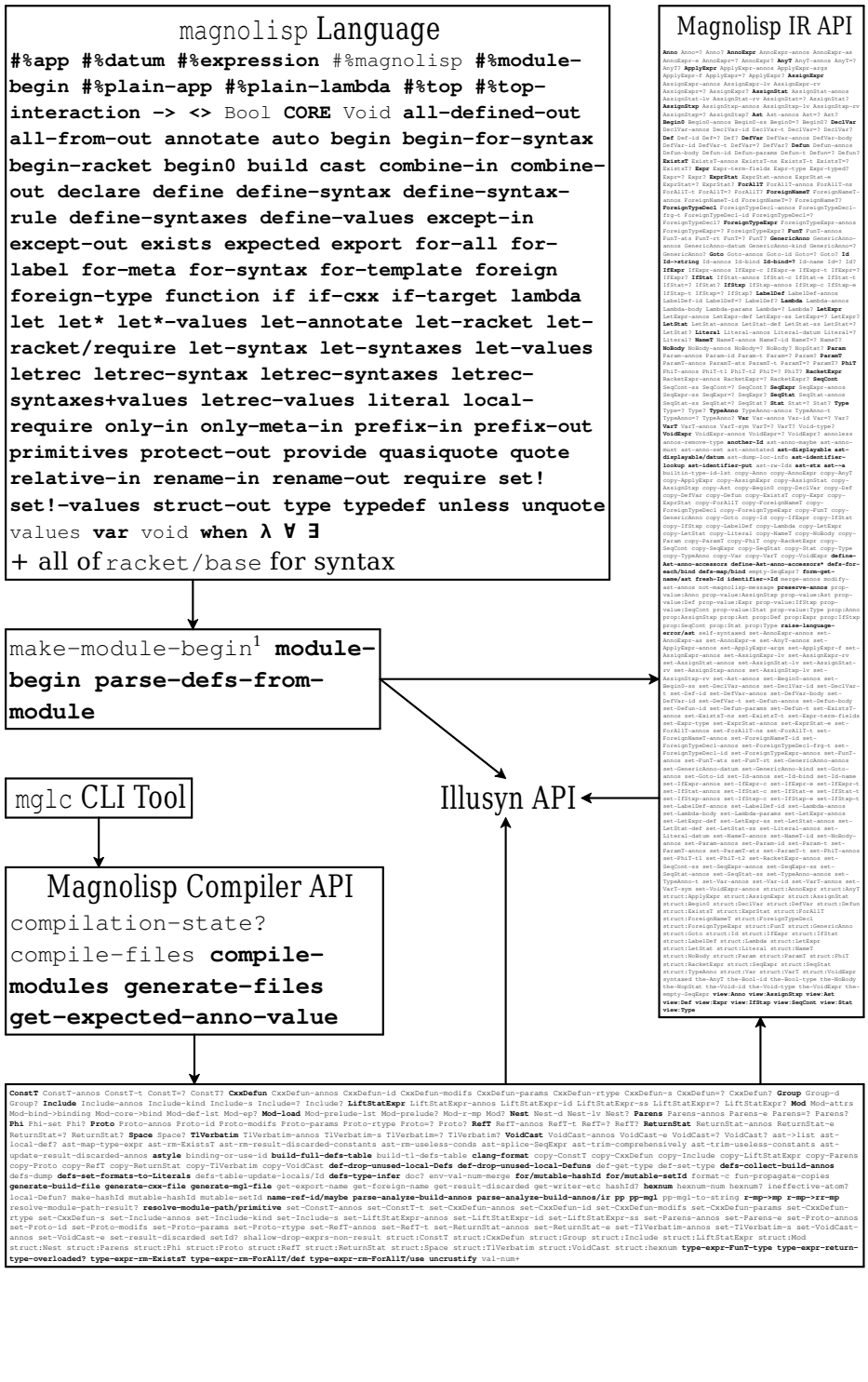
`https://bldl.ii.uib.no/software/pltnp/`

8.3.1 Illusyn

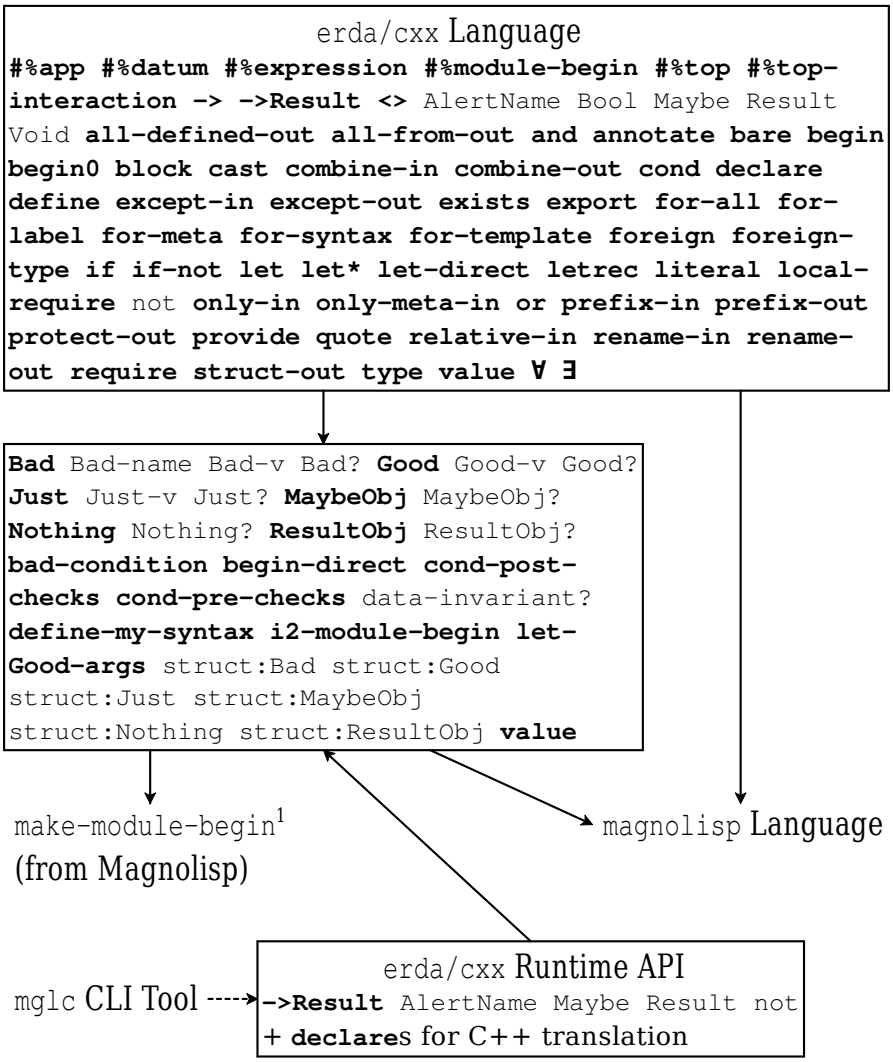
Illusyn API

<* <+ all **all-rewriter all-visitor** alltd **and-rewrite** ast-get-fields bottomup **bottomup-rewriter bottomup-visitor break** combined-rewrite-all combined-rewrite-one combined-rewrite-some combined-visit-all **conc-vspec**[1] current-strategic-data-accessors **define-ast define-ast* define-specific-data-strategy* define-view define-view*** downup downup2 extend-with-implied-views[1] fail-rw **gen:strategic gen:syntactifiable** generate-view-methods[1] id-rw **innermost-rewriter** list-all-rewriter list-all-visitor list-one-rewriter list-rewrite-all list-rewrite-one list-rewrite-some list-some-rewriter list-visit-all **make-strategic-data-accessors** make-strategy make-term-rewrite-all make-term-rewrite-one make-term-rewrite-some make-term-visit-all **make-view-all make-view-one make-view-some make-view-term-rewrite-all make-view-term-rewrite-one make-view-term-rewrite-some make-view-term-visit-all** oncetd one **one-rewriter** onebu **or-rewrite outermost-rewriter rec rec-lambda** repeat repeat-rewriter rewrite-repeat **seq-break seq-visit-break** set-term-fields some **some-rewriter** somebu sometd strategic-list-accessors strategic-term-accessors **strategic/c** strategic? **struct-copy/type-ctx** syntactifiable-mkstx **syntactifiable/c** syntactifiable? term-all-rewriter term-all-visitor term-fields term-for-each term-map term-one-rewriter **term-qty**[1] term-rewrite-all term-rewrite-all/stateful term-rewrite-one term-rewrite-some term-some-rewriter term-visit-all topdown topdown-break **topdown-rewriter** topdown-visit-break **topdown-visitor** try **view-term-fields-getter view-term-fields-setter** when-rw where where-not **with-strategic-data-accessors**

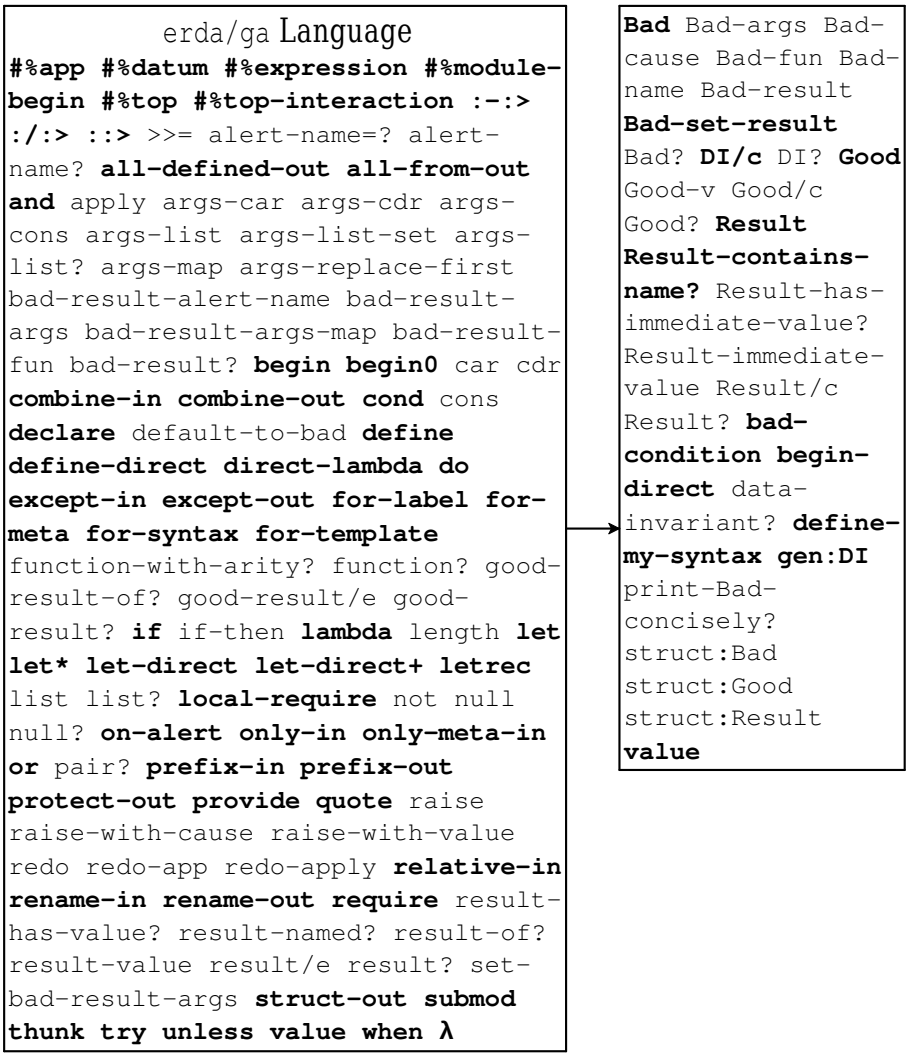
8.3.2 Magnolisp



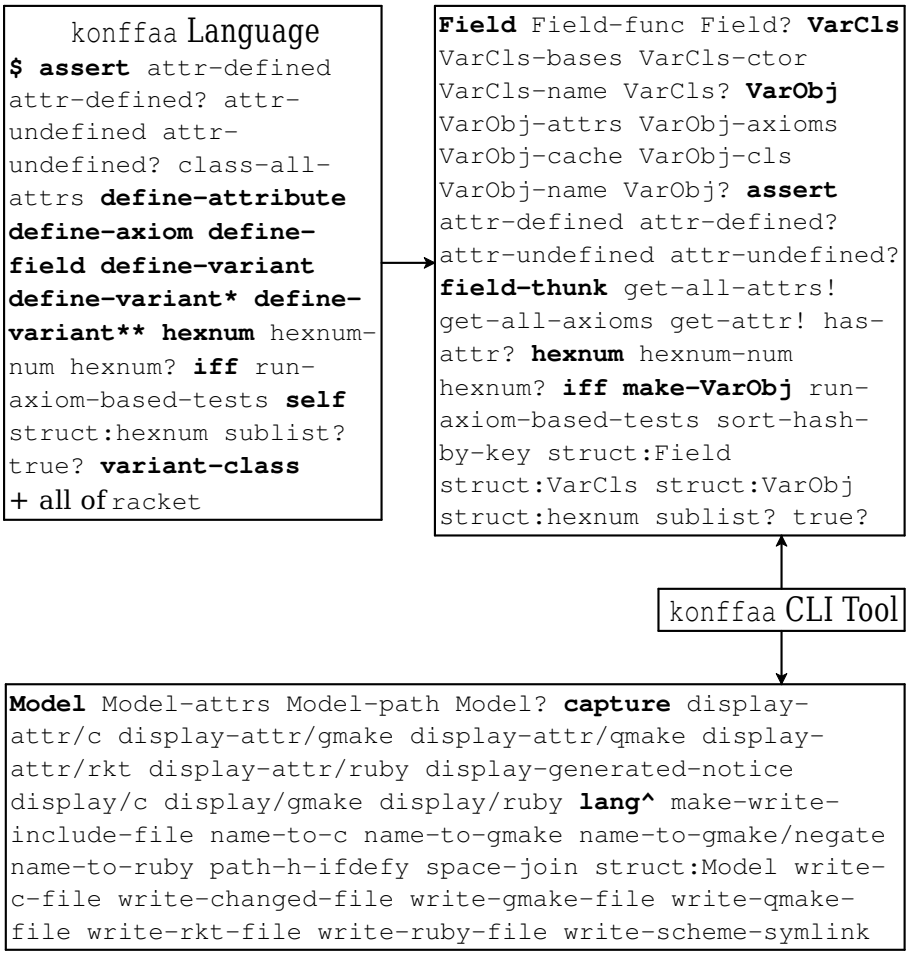
8.3.3 Erda$_{C++}$


erda/cxx Language
#%app #%datum #%expression #%module-begin #%top #%top-interaction -> ->Result <> AlertName Bool Maybe Result Void all-defined-out all-from-out and annotate bare begin begin0 block cast combine-in combine-out cond declare define except-in except-out exists export for-all for-label for-meta for-syntax for-template foreign foreign-type if if-not let let* let-direct letrec literal local-require not only-in only-meta-in or prefix-in prefix-out protect-out provide quote relative-in rename-in rename-out require struct-out type value ∀ ∃
Bad Bad-name Bad-v Bad? Good Good-v Good?
Just Just-v Just? MaybeObj MaybeObj?
Nothing Nothing? ResultObj ResultObj?
bad-condition begin-direct cond-post-checks cond-pre-checks data-invariant?
define-my-syntax i2-module-begin let-Good-args struct:Bad struct:Good
struct:Just struct:MaybeObj
struct:Nothing struct:ResultObj value
make-module-begin[1]
(from Magnolisp)
magnolisp Language
mglc CLI Tool
erda/cxx Runtime API
->Result AlertName Maybe Result not
+ declares for C++ translation

8.3.4 Erda$_{GA}$


erda/ga Language
#%app #%datum #%expression #%module-begin #%top #%top-interaction :-:> :/:> ::> >>= alert-name=? alert-name? all-defined-out all-from-out and apply args-car args-cdr args-cons args-list args-list-set args-list? args-map args-replace-first bad-result-alert-name bad-result-args bad-result-args-map bad-result-fun bad-result? begin begin0 car cdr combine-in combine-out cond cons declare default-to-bad define define-direct direct-lambda do except-in except-out for-label for-meta for-syntax for-template function-with-arity? function? good-result-of? good-result/e good-result? if if-then lambda length let let* let-direct let-direct+ letrec list list? local-require not null null? on-alert only-in only-meta-in or pair? prefix-in prefix-out protect-out provide quote raise raise-with-cause raise-with-value redo redo-app redo-apply relative-in rename-in rename-out require result-has-value? result-named? result-of? result-value result/e result? set-bad-result-args struct-out submod thunk try unless value when λ
Bad Bad-args Bad-cause Bad-fun Bad-name Bad-result Bad-set-result Bad? DI/c DI? Good Good-v Good/c Good? Result Result-contains-name? Result-has-immediate-value? Result-immediate-value Result/c Result? bad-condition begin-direct data-invariant? define-my-syntax gen:DI print-Bad-concisely? struct:Bad struct:Good struct:Result value

8.3.5 Konffaa


konffaa Language
$ assert attr-defined attr-defined? attr-undefined attr-undefined? class-all-attrs define-attribute define-axiom define-field define-variant define-variant* define-variant** hexnum hexnum-num hexnum? iff run-axiom-based-tests self struct:hexnum sublist? true? variant-class
+ all of racket
Field Field-func Field? VarCls VarCls-bases VarCls-ctor VarCls-name VarCls? VarObj VarObj-attrs VarObj-axioms VarObj-cache VarObj-cls VarObj-name VarObj? assert attr-defined attr-defined? attr-undefined attr-undefined? field-thunk get-all-attrs! get-all-axioms get-attr! has-attr? hexnum hexnum-num hexnum? iff make-VarObj run-axiom-based-tests sort-hash-by-key struct:Field struct:VarCls struct:VarObj struct:hexnum sublist? true?
konffaa CLI Tool
Model Model-attrs Model-path Model? capture display-attr/c display-attr/gmake display-attr/qmake display-attr/rkt display-attr/ruby display-generated-notice display/c display/gmake display/ruby lang^ make-write-include-file name-to-c name-to-gmake name-to-gmake/negate name-to-ruby path-h-ifdefy space-join struct:Model write-c-file write-changed-file write-gmake-file write-qmake-file write-rkt-file write-ruby-file write-scheme-symlink

Bibliography

[1] Michael D. Adams. Towards the essence of hygiene. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2015.

[2] Eric Allen, Ryan Culpepper, Danus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a syntax. In *Proceedings of International Workshop on Foundations of Object-Oriented Languages (FOOL'09)*, 2009.

[3] Android Open Source Project. Android Developers. URL `https://developer.android.com/`. Retrieved May 2013.

[4] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, Phillipa Gill, and David Lie. Short paper: A look at smartphone permission models. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 63–68, 2011.

[5] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: analyzing the Android permission specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, 2012.

[6] Anya Helene Bagge. *Constructs & Concepts: Language Design for Flexibility and Reliability*. PhD thesis, Research School in Information and Communication Technology, Department of Informatics, University of Bergen, Norway, 2009.

[7] Anya Helene Bagge. Language description for frontend implementation. In Claus Brabrand and Pierre-Etienne Moreau, editors, *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, LDTA '10, pages 9:1–9:8, New York, NY, USA, November 2010a. ACM.

[8] Anya Helene Bagge. Yet another language extension scheme. In Mark van den Brand, Dragan Gašević, and Jeff Gray, editors, *SLE '09: Proceedings of the Second International Conference on Software Language Engineering*, volume 5969 of *LNCS*, pages 123–132. Springer-Verlag, March 2010b.

[9] Anya Helene Bagge. Separating exceptional concerns. In *Proceedings of the 5th International Workshop on Exception Handling (WEH'12)*, pages 49–51. IEEE, June 2012.

[10] Anya Helene Bagge. Facts, resources and the IDE/compiler mind-meld. In *Proceedings of the 4th International Workshop on Academic Software Development Tools and Techniques (WASDeTT'13)*, July 2013. URL `http://wasdett.org/2013/submissions/wasdett2013_submission_10.pdf`.

[11] Anya Helene Bagge and Tero Hasu. A pretty good formatting pipeline. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Proceedings of the 6th International Conference on Software Language Engineering*, volume 8225 of *LNCS*, pages 177–196. Springer-Verlag, October 2013.

[12] Anya Helene Bagge and Magne Haveraaen. Axiom-based transformations: Optimisation and testing. In Jurgen J. Vinju and Adrian Johnstone, editors, *Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, volume 238 of *Electronic Notes in Theoretical Computer Science*, pages 17–33, Budapest, Hungary, 2009. Elsevier.

[13] Anya Helene Bagge and Magne Haveraaen. Interfacing concepts: Why declaration style shouldn't matter. In Torbjörn Ekman and Jurgen J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 37–50, York, UK, 2010. Elsevier.

[14] Anya Helene Bagge and Magne Haveraaen. Programming by concept. Unpublished manuscript, 2013.

[15] Anya Helene Bagge and Magne Haveraaen. Specification of generic APIs, or: Why algebraic may be better than pre/post. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 71–80, New York, NY, USA, 2014. ACM.

[16] Anya Helene Bagge, Valentin David, Magne Haveraaen, and Karl Trygve Kalleberg. Stayin' alert: Moulding failure and exceptions to your needs. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*, pages 265–274, Portland, Oregon, October 2006. ACM Press.

[17] Anya Helene Bagge, Valentin David, and Magne Haveraaen. The axioms strike back: Testing with concepts and axioms in C++. In *GPCE '09: Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, pages 15–24, New York, NY, USA, 2009. ACM.

[18] Eli Barzilay, Ryan Culpepper, and Matthew Flatt. Keeping it clean with syntax parameters. In *Proc. Workshop on Scheme and Functional Programming*, Portland, Oregon, 2011.

[19] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6): 355–371, 2004.

[20] Don S. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of Software Product Line Conference (SPLC)*, pages 7–20, September 2005.

[21] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: A tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 22–34, New York, NY, USA, 2015. ACM.

[22] David Benavides, Sergio Segura, and Antonio Ruiz-Cort'es. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010.

[23] Bergen Language Design Laboratory. Anyxporter. URL `https://github.com/bldl/anyxporter`.

[24] Bergen Language Design Laboratory. The Magnolia programming language, 2013. `http://magnolia-lang.org/`.

[25] BlackBerry. BlackBerry Developer. URL `http://developer.blackberry.com/`. Retrieved May 2013.

[26] BlackBerry. BlackBerry 10 Native SDK 10.0.9. Software distribution, December 2012.

[27] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 1–12, New York, NY, USA, 2009. ACM.

[28] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[29] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 195–204. IEEE Computer Society Press, 1981.

[30] Ali Çehreli. *Programming in D*. Ali Çehreli, 2016. Revision: 2016-04-06.

[31] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP)*, January 2011.

[32] Hans Chalupsky and Robert M. MacGregor. STELLA - a Lisp-like language for symbolic programming with delivery in Common Lisp, C++ and Java. In *Proceedings of Lisp User Group Meeting*, 1999.

[33] Philippe Charles, Robert M. Fuhrer, Stanley M. Sutton, Jr., Evelyn Duesterwald, and Jurgen 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, pages 191–206, New York, NY, USA, 2009. ACM.

[34] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable components of semantic specifications. In *Transactions on Aspect-Oriented Software Development XII*, volume 8989 of *Lecture Notes in Computer Science*, pages 132–179. Springer, 2015.

[35] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 155–162, New York, NY, USA, 1991. ACM.

[36] ClojureScript, 2016. URL `https://github.com/clojure/clojurescript`.

[37] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[38] William R. Cook. On understanding data abstraction, revisited. In *Proceedings of Onward! Essays*, pages 557–572, 2009.

[39] Ryan Culpepper. Fortifying macros. *Journal of Functional Programming*, 22:439–476, September 2012.

[40] Ryan Culpepper and Matthias Felleisen. A stepper for Scheme macros. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*, September 2006.

[41] Ryan Culpepper and Matthias Felleisen. Debugging macros. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 135–144, New York, NY, USA, 2007. ACM.

[42] Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, GPCE'05, pages 373–388, Berlin, Heidelberg, 2005. Springer-Verlag.

[43] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 411–422, New York, NY, USA, June 2013. ACM Press.

[44] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22 (3):183–195, 1994.

[45] Merijn de Jonge. Build-level components. *IEEE Trans. Software Eng.*, 31 (7):588–600, 2005.

[46] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[47] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A multi-stage language for high-performance computing. *SIGPLAN Not.*, 48(6):105–116, June 2013.

[48] Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript: Hygienic macros for ES5. In *Proceedings of the Dynamic Languages Symposium*, October 2014.

[49] Eelco Dolstra. Integrating software construction and software deployment. In Bernhard Westfechtel and André van der Hoek, editors, *Proceedings of 11th International Workshop on Software Configuration Management (SCM-11)*, volume 2649 of *Lecture Notes in Computer Science*, pages 102–117, May 2003.

[50] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp Symb. Comput.*, 5(4):295–326, 1992.

[51] Carl Eastlund. *Modular Proof Development in ACL2*. PhD thesis, Northeastern University, 2012.

[52] Carl Eastlund and Matthias Felleisen. Hygienic macros for ACL2. In *Proceedings of Trends in Functional Programming (TFP)*, 2010.

[53] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNAI*, pages 273–298, 2007.

[54] Sebastian Erdweg and Felix Rieger. A framework for extensible languages. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 3–12, New York, NY, USA, 2013. ACM.

[55] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In *Proceedings of the 2011 ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '11, pages 391–406, New York, NY, USA, 2011. ACM.

[56] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of the 12th Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 1–8, New York, NY, USA, 2012. ACM.

[57] Stuart I. Feldman. Make – a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[58] Matthias Felleisen, Mitch Wand, Daniel Friedman, and Bruce Duba. Abstract continuations: A mathematical semantics for handling full jumps. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 52–62, New York, NY, USA, 1988. ACM.

[59] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket manifesto. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–128, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[60] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 180–190, New York, NY, USA, 1988. ACM.

[61] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, 2011.

[62] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, 2012.

[63] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, March 2002.

[64] David Fisher and Olin Shivers. Building language towers with Ziggurat. *Journal of Functional Programming*, 18(5-6):707–780, September 2008.

[65] Matthew Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, Houston, Texas, June 1999.

[66] Matthew Flatt. Composable and compilable macros: You want it when? In *International Conference on Functional Programming (ICFP)*, pages 72–83, October 2002.

[67] Matthew Flatt. Submodules in Racket: You want it when, again? In *12th International Conference on Generative Programming (GPCE '13)*. ACM, October 2013.

[68] Matthew Flatt. Binding as sets of scopes. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2015.

[69] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 236–248, New York, NY, USA, 1998. ACM.

[70] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[71] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *International Conference on Functional Programming (ICFP 2007)*, pages 165–176, October 2007.

[72] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the book on ad hoc documentation tools. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*, pages 109–120, August 2009.

[73] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that work together: Compile-time bindings, partial expansion, and definition contexts. *J. Funct. Program.*, 22(2):181–216, March 2012.

[74] John D. Gannon, Paul R. McMullin, and Richard G. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981.

[75] Blaine Garst. Apple's extensions to C. Technical Report N1370, WG14, March 2009.

[76] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 313–323, New York, NY, USA, 1998. ACM.

[77] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5):1–11, May 2003.

[78] J. A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, SE-10(5):528–543, September 1984.

[79] J.A. Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, 1986.

[80] Joseph A. Goguen and Rod M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[81] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 291–310, New York, NY, USA, 2006. ACM.

[82] James W. Grenning. *Test-Driven Development for Embedded C*. The Pragmatic Programmers, LLC, April 2011.

[83] Maarit Harsu. A survey on domain engineering. Technical Report 31, Institute of Software Systems, Tampere University of Technology, 2002.

[84] Tero Hasu. ContextLogger2—a tool for smartphone data gathering. Technical Report 2010-1, Helsinki Institute for Information Technology HIIT, Aalto University, August 2010.

[85] Tero Hasu. Concrete error handling mechanisms should be configurable. In *Proceedings of the 5th International Workshop on Exception Handling (WEH'12)*, pages 46–48. IEEE, June 2012.

[86] Tero Hasu. Managing language variability in source-to-source compilers by transforming illusionary syntax. In *Proceedings of the 2nd International Workshop on Open and Original Problems in Software Language Engineering (OOPSLE 2014)*, pages 11–14, February 2014.

[87] Tero Hasu and Matthew Flatt. Source-to-source compilation in Racket: You want it in which language? In *Preproceedings of the 26nd Symposium on Implementation and Application of Functional Languages (IFL 2014)*, October 2014.

[88] Tero Hasu and Matthew Flatt. Source-to-source compilation via submodules. In *Proceedings of the 9th European Lisp Symposium (ELS 2016)*, May 2016.

[89] Tero Hasu and Magne Haveraaen. Errors as data values as the language default. In *Proceedings of 27th Nordic Workshop on Programming Theory (NWPT 2015)*, October 2015.

[90] Tero Hasu and Magne Haveraaen. Errors as data values. In *Proceedings of the Norwegian Informatics Conference (NIK)*, November 2016. To appear.

[91] Tero Hasu, Anya Helene Bagge, and Magne Haveraaen. Inferring required permissions for statically composed programs. In H. Riis Nielson and D. Gollmann, editors, *NordSec 2013*, volume 8208 of *Lecture Notes in Computer Science*, pages 51–66, Berlin, Heidelberg, October 2013. Springer-Verlag.

[92] Magne Haveraaen and Eric G. Wagner. Guarded algebras: Disguising partiality so you won't know whether it's there. In *Recent Trends In Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 3–11. Springer-Verlag, 2000.

[93] Craig Heath. *Symbian OS Platform Security: Software Development Using the Symbian OS Security Architecture*. Wiley, February 2006.

[94] David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 16–27. ACM Press, September 2004.

[95] David Hernie. Windows Phone 8 security deep dive. Slide set, October 2012.

[96] William A. Hetrick, Charles W. Krueger, and Joseph G. Moore. Incremental return on incremental investment: Engenio's transition to software product line practice. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '06, pages 798–804, New York, NY, USA, 2006. ACM.

[97] Tasuku Hiraishi, Masahiro Yasugi, and Taiichi Yuasa. Experience with SC: Transformation-based implementation of various language extensions to C. In *International Lisp Conference 2007*, pages 103–113, Cambridge, UK, 2007.

[98] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer Berlin Heidelberg, 1991.

[99] IEEE. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754™-2008)*. IEEE Computer Society, August 2008.

[100] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, July 2005.

[101] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 145–158, New York, NY, USA, 2013. ACM.

[102] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[103] Antti Kantee and Heikki Vuolteenaho. Experiences in portable mobile application development. In *Proceedings of First International Workshop on Advanced Software Engineering: Expanding the Frontiers of Software Technology*, Santiago, Chile, August 2006.

[104] Deepak Kapur, David R. Musser, and Alexander A. Stepanov. Tecton: A language for manipulating generic objects. In J. Staunstrup, editor, *Proceedings of a Workshop on Program Specification*, Lecture Notes in Computer Science, pages 402–414, Aarhus, Denmark, August 1981. Springer-Verlag.

[105] Christian Kästner, Sven Apel, and Klaus Ostermann. The road to feature modularity? In *Proceedings of the 3rd International Workshop on Feature-Oriented Software Development (FOSD)*, pages 5:1–5:8, New York, NY, September 2011. ACM Press.

[106] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 773–792, New York, NY, October 2012. ACM Press.

[107] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, pages 444–463, October 2010.

[108] Andrew Kelley. Introduction to the Zig programming language, February 2016. URL `http://andrewkelley.me/post/intro-to-zig.html`.

[109] Richard Kelsey, William Clinger, Jonathan Rees, Hal Abelson, N. I. Adams IV, R. Kent Dybvig, Christopher T. Haynes, Guillermo J. Rozas, Daniel P. Friedman, D. H. Bartley, R. Halstead, D. Oxley, Eugene Kohlbecker, G. J. Sussman, G. Brooks, Chris Hanson, Guy L.. Steele, Jr, Kent M. Pitman, and Mitchell Wand. Revised^5 report on the algorithmic language Scheme. *SIGPLAN Not.*, 33(9):26–76, 1998.

[110] Oleg Kiselyov, Simon Peyton-Jones, and Amr Sabry. Lazy v. yield: Incremental, linear pretty-printing. In *10th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 190–206, December 2012.

[111] Paul Klint, Tijs van der Storm, and Jurgen 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*, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society.

[112] Donald E. Knuth. Semantics of context-free languages. *Journal of Mathematical System Theory*, 2(2):127–145, 1968.

[113] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM.

[114] Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '87)*, 1987.

[115] Kari Kostiainen, Elena Reshetova, Jan-Erik Ekberg, and N. Asokan. Old, new, borrowed, blue – a perspective on the evolution of mobile platform security architectures. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy*, CODASPY '11, pages 13–24, 2011.

[116] Evgenii Kotelnikov. Type-directed language extension for effectful computations. In *Proceedings of the Fifth Annual Scala Workshop*, SCALA '14, pages 35–43, New York, NY, USA, 2014. ACM.

[117] Beyongcheol Lee, Robert Grimm, Martin Hirzel, and Kathryn S. McKinley. Marco: Safe, expressive macros for any language. In *European Conference on Object Oriented Programming (ECOOP)*, June 2012.

[118] Shuying Liang, Matthew Might, Thomas Gilray, and David Van Horn. Pushdown exception-flow analysis of object-oriented programs, February 2013. URL `http://arxiv.org/abs/1302.2692`.

[119] Alexander Lier, Linus Franke, Marc Stamminger, and Kai Selgrad. A case study in implementation-space exploration. In *Proceedings of the 9th European Lisp Symposium (ELS 2016)*, May 2016.

[120] Linj, 2013. URL `https://github.com/xach/linj`.

[121] Jed Liu and Andrew C. Myers. JMatch: Iterable abstract pattern matching for Java. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL'03)*, pages 110–127, January 2003.

[122] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. Wiley, November 1996.

[123] Geoffrey Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 73–82, New York, NY, USA, 2007. ACM.

[124] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[125] Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2013.

[126] Microsoft. Microsoft Developer Network. URL `http://msdn.microsoft.com/`. Retrieved Jul 2013.

[127] Zaur Molotnikov, Markus Völter, and Daniel Ratiu. Automated domain-specific C verification with mbeddr. In *Proceedings of the International Conference on Automated Software Engineering (ASE 2014)*, 2014.

[128] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *Proceedings of the International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76, 2003.

[129] Ben Morris, editor. *Introduction to bada – A Developer's Guide*. Wiley, 2010.

[130] mssf-team. Mobile Simplified Security Framework, May 2012. URL `http://gitorious.org/meego-platform-security`.

[131] Nokia Corporation. Nokia Developer. URL `http://www.developer.nokia.com/`. Retrieved May 2013.

[132] Nokia Corporation. Qt Mobility 1.2: Qt Mobility project reference documentation, 2011. URL `http://doc.qt.digia.com/qtmobility/`.

[133] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the systematic construction of generators for performance libraries. In *International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 125–134, October 2013.

[134] Chris Okasaki. Views for Standard ML. In *In SIGPLAN Workshop on ML*, pages 14–23, 1998.

[135] Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *Proceedings of International Conference on Functional Programming (ICFP 2006)*, pages 87–98, September 2006.

[136] John Pagonis. Architecture, paradigms, idioms and weirdness of the C++ in your pocket! A slideset presented at ACCU Conference, April 2007.

[137] Parenscript, 2016. URL `https://common-lisp.net/project/parenscript/`.

[138] Christian L. Petersen, Matthias Gorges, Dustin Dunsmuir, Mark Ansermino, and Guy A. Dumont. Experience report: Functional programming of mHealth applications. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 357–362, New York, NY, USA, 2013. ACM.

[139] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000.

[140] Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience report: a do-it-yourself high-assurance compiler. In *Proceedings of the Intl. Conference on Functional Programming (ICFP)*. ACM, September 2012.

[141] Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: monitoring embedded systems. *Innovations in Systems and Software Engineering: Special Issue on Software Health Management*, 9(4): 235–255, 2013.

[142] Gordon Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Proceedings of the 4th International Conference Foundations of Software Science and Computation Structures (FOSSACS)*, pages 1–24, Berlin, Heidelberg, April 2001. Springer Berlin Heidelberg.

[143] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Proceedings of the 18th European European Symposium on Programming (ESOP 2009)*, pages 80–94, Berlin, Heidelberg, March 2009. Springer Berlin Heidelberg.

[144] Justin Pombrio and Shriram Krishnamurthi. Hygienic resugaring of compositional desugaring. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 75–87, New York, NY, USA, 2015. ACM.

[145] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Berlin Heidelberg, 1997.

[146] Qt Wiki. Coding conventions, January 2016. URL `https://wiki.qt.io/Coding_Conventions`.

[147] Jon Rafkind and Matthew Flatt. Honu: Syntactic extension for algebraic notation through enforestation. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 122–131, New York, NY, USA, 2012. ACM.

[148] Pedro Ramos and António Leitão. An implementation of Python for Racket. In *European Lisp Symposium*, May 2014.

[149] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'00)*, pages 347–360, 2000.

[150] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.

[151] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th Symposium on Principles of Programming Languages (POPL '13)*, January 2012.

[152] Jane Sales, editor. *Symbian OS Internals*. Wiley, 2005.

[153] Samsung. bada Developers. URL `http://developer.bada.com`. Retrieved Mar 2013.

[154] Samsung. bada SDK 2.0.0. Software distribution, August 2011.

[155] Kai Selgrad, Alexander Lier, Markus Wittmann, Daniel Lohmann, and Marc Stamminger. Defmacro for C: Lightweight, ad hoc code generation. In *European Lisp Symposium*, May 2014.

[156] Kai Selgrad, Alexander Lier, Jan Dörntlein, Oliver Reiche, and Marc Stamminger. A high-performance image processing DSL for heterogeneous architectures. In *Proceedings of the 9th European Lisp Symposium (ELS 2016)*, May 2016.

[157] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 1–16, New York, NY, USA, 2002. ACM.

[158] Anthony Sloane. Lightweight language processing in Kiama. In *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 2011.

[159] Anthony M. Sloane. Experiences with domain-specific language embedding in Scala. In *Proceedings of the 2nd International Workshop on Domain-Specific Program Development*, 2008.

[160] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.

[161] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, 3rd edition, 1997.

[162] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the International Conference on Machine Learning*, 2011.

[163] Sagar Sunkle, Marko Rosenmüller, Norbert Siegmund, Syed Saif ur Rahman, Gunter Saake, and Sven Apel. Features as first-class entities—toward a better representation of features. In *Proceedings of Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*, pages 27–34, October 2008.

[164] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ML. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, pages 15–27, September 2011.

[165] Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ACM, October 2007.

[166] Tizen Project. Tizen Developers dev guide. URL `https://developer.tizen.org/`. Retrieved May 2013.

[167] Tizen Project. Tizen SDK 2.0. Software distribution, February 2013.

[168] Sam Tobin-Hochstadt. Extensible pattern matching in an extensible language. *CoRR*, abs/1106.2578, 2011.

[169] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. *SIGPLAN Not.*, 47 (6):132–141, June 2011.

[170] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM.

[171] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, February 1997.

[172] Federico Tomassetti and Daniel Ratiu. Extracting variability from C and lifting it to mbeddr. In *Proceedings of REVE 2013 Workshop*, 2013.

[173] David Turner. Robust design techniques for C programs, 2014. URL `http://freetype.sourceforge.net/david/reliable-c.html`. Draft paper, retrieved in August 2014.

[174] UBM Electronics. 2012 Embedded Market Survey, 2012.

[175] L. Thomas van Binsbergen, Neil Sculthorpe, and Peter D. Mosses. Tool support for component-based semantics. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 8–11. ACM, 2016.

[176] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, March 2000.

[177] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th International Conf. on Compiler Construction (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[178] T. Vidas, N. Christin, and L. Cranor. Curbing Android permission creep. In *Proceedings of the Web 2.0 Security and Privacy 2011 Workshop (W2SP 2011)*, Oakland, CA, May 2011.

[179] Eelco Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.

[180] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA '01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[181] Markus Voelter. *Generic Tools, Specific Languages*. PhD thesis, Delft University of Technology, 2014.

[182] Markus Voelter and Sascha Lisson. Supporting diverse notations with MPS' projectional editor. In *Proceedings of GEMOC 2014 Workshop*, 2014.

[183] Markus Voelter and Eelco Visser. Product line engineering using domain-specific languages. In *Proceedings of the 15th International Software Product Line Conference (SPLC)*, 2011.

[184] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an extensible C-based programming language and IDE for embedded systems. In *Proceedings of SPLASH Wavefront 2012*, 2012.

[185] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):1–52, 2013a.

[186] Markus Voelter, Daniel Ratiu, and Federico Tomassetti. Requirements as first-class citizens. In *Proc. Modellbasierte Entwicklung eingebetteter Systeme IX, MBEES '13, Schloss Dagstuhl*, 2013b.

[187] Markus Voelter, Arie van Deursen, Bernd Kolb, and Stephan Eberle. Using C language extensions for developing embedded software: A case study. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 655–674, New York, NY, USA, October 2015. ACM.

[188] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Principles of Programming Languages*, 1999.

[189] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, 1987.

[190] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer-Verlag, 1995.

[191] Elecia White. *Making Embedded Systems: Design Patterns for Great Software*. O'Reilly Media, November 2011.

[192] Kevin Williams, Matt Le, Ted Kaminski, and Eric Van Wyk. A compiler extension for parallel matrix programming. In *International Conference on Parallel Processing (ICPP-2014)*, September 2014.

[193] David W. Wood. *Smartphones and Beyond: Lessons from the remarkable rise and fall of Symbian*. David W. Wood, 2014.

[194] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2):39–54, 2010.

[195] Danny Yoo and Shriram Krishnamurthi. Whalesong: Running Racket in the browser. In *Proceedings of the 9th Dynamic Languages Symposium (DLS 2013)*, pages 97–108, New York, NY, USA, October 2013. ACM.

Citation Index

Haveraaen and Wagner [2000], 32, 109, 112
Haveraaen and Wagner [92], 184
Heath [93], 93, 184
Herman and Meunier [2004], 23
Herman and Meunier [94], 184
Hernie [95], 94, 184
Hetrick et al. [2006], 8
Hetrick et al. [96], 185
Hiraishi et al. [2007], 55, 149
Hiraishi et al. [97], 185
Hölzle et al. [98], 82, 185
IEEE [2008], 110, 127
IEEE [99], 185
Ierusalimschy et al. [100], 185
Ierusalimschy et al. [2005], 29
Kammar et al. [101], 185
Kammar et al. [2013], 128
Kang et al. [102], 185
Kang et al. [1990], 8
Kantee and Vuolteenaho [103], 185
Kantee and Vuolteenaho [2006], 27
Kapur et al. [104], 185
Kapur et al. [1981], 13, 14
Kästner et al. [105], 185
Kästner et al. [106], 185
Kästner et al. [2011], 7
Kästner et al. [2012], 15
Kats and Visser [107], 70, 186
Kats and Visser [2010], 24, 51, 164
Kelley [108], 186
Kelley [2016], 127, 144
Kelsey et al. [109], 186
Kelsey et al. [1998], 20, 21
Kiselyov et al. [110], 186
Kiselyov et al. [2012], 161
Klint et al. [111], 70, 186
Klint et al. [2009], 51
Knuth [112], 186
Knuth [1968], 2
Kohlbecker and Wand [114], 186
Kohlbecker and Wand [1987], 20
Kohlbecker et al. [113], 186
Kohlbecker et al. [1986], 3, 20
Kostiainen et al. [115], 90, 92, 186
Kotelnikov [116], 186
Kotelnikov [2014], 126, 128
Lee et al. [117], 79, 187
Liang et al. [118], 187
Liang et al. [2013], 108
Lier et al. [119], 187
Lier et al. [2016], 162
Linj [120], 187
Linj [2013], 53
Liu and Myers [121], 78, 85, 187
Loeckx et al. [122], 187
Loeckx et al. [1996], 17
Mainland [123], 187
Mainland [2007], 128
McBride and Paterson [124], 187
McBride and Paterson [2008], 128
McDonell et al. [125], 187
McDonell et al. [2013], 55
Microsoft [126], 91, 94, 187
Molotnikov et al. [127], 187
Molotnikov et al. [2014], 165
Moreau et al. [128], 77, 187
Morris [129], 187
Morris [2010], 9
mssf-team [130], 93, 187
Nokia Corporation [131], 93, 188
Nokia Corporation [132], 98, 188
Ofenbeck et al. [133], 188
Ofenbeck et al. [2013], 54
Okasaki [134], 73, 77, 188
Owens and Flatt [135], 188
Owens and Flatt [2006], 51
Pagonis [136], 188
Pagonis [2007], 27
Parenscript [137], 188
Parenscript [2016], 53
Petersen et al. [138], 188
Petersen et al. [2013], 162
Pierce and Turner [139], 188
Pierce and Turner [2000], 147
Pike et al. [140], 188
Pike et al. [141], 188
Pike et al. [2012], 55
Pike et al. [2013], 55
Plotkin and Power [142], 188
Plotkin and Power [2001], 128
Plotkin and Pretnar [143], 188
Plotkin and Pretnar [2009], 128
Pombrio and Krishnamurthi [144], 189
Pombrio and Krishnamurthi [2015], 25
Prehofer [145], 189
Prehofer [1997], 7
Qt Wiki [146], 189

Index