Domain-specific languages in software development

and the relation to partial evaluation

Niels H. Christensen

Preface

The present document is the Ph.D. thesis of Niels H. Christensen. The thesis is the main result of the author's work on an industrial Ph.D. project, which has been performed in collaboration between DIKU (Department of Computer Science, Faculty of Science, University of Copenhagen), Danish IT company Visanti A/S and the Danish Academy of Technical Sciences (ATV). Professor Neil D. Jones has been the academic supervisor, while Finn Normann Pedersen has been the industrial supervisor. Professor Christian S. Jensen of Aalborg University has been ATV's representative on the project. The author has also written a companion *industrial report* (erhvervsrapport) for the project. That report is confidential.

This thesis has been submitted to the University of Copenhagen on the 1st of July 2003. One chapter is joint work with another author. As the rules of the university demand, the submitted thesis was accompanied by a statement from that author describing the distribution of responsibilities in the joint work.

Compared to most theses and papers written in LaTeX, this document is set in an unusual font. The font was produced by using the `pslatex` command and setting font size to 12 pt. Hopefully, the reader will agree that what this font lacks in "scientific look", it makes up for in readability.

Contents

1 - Introduction

Beware of the Turing tar-pit
in which everything is possible
but nothing of interest is easy.
Alan J. Perlis

This thesis is about *domain-specific languages*, a particular breed of programming languages, each one tailored to solve problems of a specific kind.

My first domain-specific language was developed in 1989, I think. I was programming *Mallard BASIC* on the Amstrad PCW 8256 as a hobby. Many of my programs would print out a full page of text, either as introduction to the user when the program was started or as explanatory text on demand. I liked working with the formatting of these things, getting it *exactly* the way I wanted it. Of course, getting it right by continously editing two dozen `PRINT` statements and rerunning the program is not very efficient, but I guess it was part of the fun – at least until I came up with something even more fun!

At one point I wrote a program that would read in a text file and display it on my black/green monitor, formatted with respect to certain codes that could be inserted into the text itself. Without knowing it, I was reinventing *HTML* without the *H* – and probably doing a bad job at it, too. But my text displaying program was being directed by the codes, so I count it as my first interpreter. And the language it was interpreting was (very!) domain-specific: it was only useful for formatting text on a monochrome screen (academically speaking, it solved problems in the domain of text formatting). Little did I know that Jon Bentley had already written his *Little Languages* piece in the *Communications of the ACM* ([Ben86]).

1.1 What is a domain-specific language?

I've met a number of different views as to what actually constitutes a domain-specific language. Most will agree that it's some sort of restricted programming language, but that's where the agreement ends. Some refuse to accept something that is not Turing complete (i.e. a language that lets you compute *anything* that any other language can, see [Tur36],[Jon97]) as a programming language. This rules out my little language described above, *HTML*, core *SQL* and many others as domain-specific languages.

Other people hesitate to call a programming language domain-specific if it *is* Turing complete, as it can then arguably solve *any* problem, regardless of which problem domain you would say it belongs to. Is *Fortran* domain-specific? Sure, it's best for numerical computations but you can use it for programming anything from databases to games. And is "numerical problems" something you would accept as delimiting a problem domain?

Others seem to accept *any* programming language as domain-specific; someone told me *C* is domain-specific because it is very good for writing operating systems and other "machine-near" programs.

This – rather metaphysical – definitional debate is not of interest to the present author, and we shall not dwell on it. The focus of this thesis is on the situation where a number of more or less well-understood problems (that are to be solved by a computer) are given, and someone chooses to approach this task by *inventing* a programming language that is somehow more handy for writing the appropriate applications than the languages that already existed.

So instead of debating whether a programming language *is* domain-specific, the thesis will focus on issues relating to the language-inventing approach and try to make use of some of the wisdom and techniques that have been put forward under the banner of domain-specific languages.

What's in a word? The term "domain-specific language" is really an abbrieviation of the even less elegant "domain-specific programming language". I'm not sure who invented these names, but Bentley ([Ben86]) uses the less cumbersome but also less informative "little language". In this thesis, I will mainly use "domain-specific language", often abbrieviated to *DSL*.

1.2 Disclaimer

This thesis is not meant to be a scientific advertisement for domain-specific languages. Approaching a given task by inventing a programming language has proved itself successful a number of times but it would be ridiculous to call it a panacea. One characteristic of the approach that is often problematic, is that it generally requires you to give a meta-solution to the given problems *and many more* before you start programming for real.

In his short position paper *Zeppelins and jet planes: a metaphor for modern software projects* ([Arm01]), Phillip Armour likens software projects with the problem of military ground-to-air battle. The first attempts at fighting a war using aviation were during the first World War, when Germany bombed London using zeppelins. This spurred the need (on the British side) for shooting down these zeppelins.

"To successfully bag a zeppelin," Armour writes, "we would need to know, as precisely as possible, the following information: for the airship — altitude, distance down range, velocity, size; for the ground cannon, we'd need to know muzzle velocity, projectile air resistance, temperature/density, wind velocity, among others". Measuring all of these values, and combining them in a ballistic equation to compute the values resulting in a hit is not easy. But – and this is the point – once we know all of these values and fire our cannon appropriately, the zeppelin is history.

That whole scenario, the meticulous measurements and the firing of a ballistic missile, is of little use in modern warfare. The zeppelins have been replaced by supersonic planes, and trying to shoot down one of these in the above manner would be way too slow. What is even worse is that knowing the values listed by Armour is of no use. Should we try to fire on the predicted future position of a fast-moving jet fighter, the pilot would simply avoid our attack by maneuvering his plane in another direction, spoiling our predictions. The plane moves unpredictably *because of what we do*. That's why ground-to-air fight now involves high-speed, target-seeking missiles.

Some software development projects are like shooting down zeppelins. The task is predictable although the parameters are unknown. The project team can figure out the parameters using more or less well-known methods, and once they're all collected, finishing the job is routine.

But many projects (Armour says *today's projects*) aren't like that. The precise aim of a project is often not clear ("We need a Web GUI for our product!"), and it may continue to move as we work ("But even more importantly, the Web GUI should demonstrate features of our new product!"), and the aim may even be influenced by the fact that the project has been started ("The guys heard that our Web GUI is on the way, so they added functionality for online payment, which we should support!"). That's why, Armour argues, we need software development methods less like ballistic artillery and more like target-seeking missiles.

Most methods for designing and implementing DSLs work like ballistic artillery, see for instance [CE00, MWW02, Thi98, Wei, SeaBP99, vDKV00]. Planning steps – i.e. the steps before any production starts – may include commonality analysis, domain analysis, domain engineering, feature modelling, and more (see [CE00] for very thorough descriptions of such analyses). In some sense (not to be elaborated on here) solving problems through DSL creation may be the *ultimate* "plan first, *then* execute" approach. And in Armour's parallel, this may be a major problem with employing DSLs in a given project if it turns out to be of jet plane kind.

1.3 Motivation for the project

The work behind this thesis has its roots in a previous research project called *PEKIS*, which is a Danish acronym for *Partial Evaluation in a Complex, Industrial System*. The *PEKIS* project was instigated by the current author and ran as a collaboration between *CIT* (Danish Center of IT Research), *DIKU* (Dept. of Computer Science, University of Copenhagen) and the Danish IT company *MindPass*. We were a small group of *DIKU* students and *MindPass* employees who worked on optimizing *C* code from *MindPass*'s system using the *C-Mix* partial evaluator tool ([And94]). As it most oftens happens, some of our attempts worked out well, others didn't.

The most interesting experiment we did was also the last one, and of course we ran out of time just when things were getting exciting. The experiment used *C-Mix* to implement a domain-specific language, allowing us to *generalize* the functionality of an existing subsystem rather than optimizing it. To the reader unfamiliar with partial evaluation, the case that we worked with serves well to illustrate the link between that technique and domain-specific languages.

The particular subsystem generated database queries to find data records that *approximated* given input values. As a concrete example, imagine that each record in a database table *Shows* represents the performance of a theatre play or a musical on a Copenhagen scene. Among other attributes, each record would have a real number attribute *duration*, telling how long time the given show takes. To keep the example simple, we'll assume that potential audience looking for a show to watch would only care about the duration of the show. So you could specify to the *MindPass* system that you want to see a show with a duration of about 2 hours. A bit more or a bit less is OK, but the closer to two hours the better.

The system would translate this to a priority number which for a given duration was e.g.

$$1 - |2 - duration|$$

So a priority of 1 corresponds to a perfect match, a priority of $\frac{1}{2}$ to a reasonable match while a priority of 0 or below represents a really unattractive option. The system would compute this priority for each record, and list the shows in order, sorted by this number. But to increase performance, it would also restrict its search to somewhat reasonable matching records, say ones where the priority is positive, i.e. where $1 < duration < 3$.

So all of this calls for three functions in the system:

- *pri(optimalDur,actualDur)*, returning the priority $1 - |optimalDur - actualDur|$.
- *lower(optimalDur)*, returning the lower *duration* bound for reasonably matching records.

- *upper(optimalDur)*, returning the upper *duration* bound for reasonably matching records.

But the function $1 - |optimalDur - actualDur|$ is rather arbitrary. In other situations (or for some users), we might need to scale the absolute difference term. Or maybe a shorter duration than the optimal is far better than a longer duration. Or perhaps priority should decrease quadratically with the difference (this is only important when the duration priority is combined with other measures, of course). So in a realistic system we could end up with many versions of each of the three functions. This would be a mess, especially since *lower* and *upper* are tightly coupled to their corresponding version of *pri*.

What we did was essentially to write a version of the system where the expression for computing priority was variable; it could be defined in a little expression language. The new function for computing proirities would superficially look like this:

```
float pri(Expression e,float optimalDur,float actualDur){
  if(isLookupOptimal(e))
    return optimalDur;
  if(isLookupActual(e))
    return actualDur;
  if(isAbs(e)){
    float x = eval(e.subExp);
    return (x>0) ? x : -x;
  };
  if(isMinus(e)){
    float x = pri(e.leftSubExp,optimalDur,actualDur);
    float y = pri(e.RightSubExp,optimalDur,actualDur);
    return (x-y);
  //...and many more cases for e
}
```

The functions for *lower* and *upper* are more complex as they have to perform symbolic constraint solving with constraints derived from an input `Expression e`. As can be seen, while the new versions are far more versatile than the original versions, they are also far less efficient. The performance of the system will be subject to *interpretive overhead*, interpreting our little expression language.

But of course we only allowed the introduction of the disease because we knew we had the cure: partial evalution. A partial evaluator like *C-Mix* can read in a function like the above along with an input value *v*, and it will output a specialized version of the function that has one less parameter – the first parameter has been fixed to the value *v*. But – and this is the point – it may also have removed any interpretive overhead associated with the fixed parameter!

More concretely, if we feed *C-Mix* with the function *pri* above and an input value representing the expression $1 - |optimalDur - actualDur|$, it will output something similiar to this:

```
float pri(float optimalDur,float actualDur){
  float x1 = optimalDur - actualDur;
  float x2 = (x1 > 0) ? x1 : -x1;
  return 1 - x2;
}
```

So we get the original version, with original efficiency. But in the process, we have made it possible to generate many different versions of the system, without the mess of programmers having to rewrite tightly coupled functions, and without interpretive overhead.

As mentioned, the results we got from our *PEKIS* experiments were quite encouraging. This led *MindPass* and me to apply for an industrial Ph.D. stipend to investigate further aspects of partial evaluation, and of using domain-specific languages in an industrial setting.

1.4 How the project evolved

The Ph.D. project behind this thesis was initiated July 2000 and was sponsored by a grant from the Danish Academy of Technical Sciences (*ATV*). The grant originated from the Danish Ministry of Commerce and paid tuition for the University of Copenhagen, travel expenses and part of my wages from the industrial partner.

Although the example described above had inspired the application, *MindPass* and I were interested in finding other modules in their software that could benefit from the creation of a domain-specific language. We found two such modules, and I started working with the first of these. Unfortunately, by the end of 2000 I had to change my focus as the related activities in *MindPass* were downsized. The work on the second module then developed through the spring of 2001, and progress was very satisfactory. The first application project had been badly coordinated with the *MindPass* engineers, but the new project had much better organizational support. We had an interpreter for my language integrated with the rest of the system by April 2001.

But in May 2001 my project came to a dead halt as *MindPass* declared bankruptcy, leaving the author unemployed and the project discontinued. The only way to salvage my Ph.D. project was to find a new industrial partner. During the summer I was in contact with the future founders of *Visanti*, a new Danish IT company. Our discussions lent some hope to the possibility of continuing the project, but my situation made it hard to make any kind of progress. I decided to go through with my planned visit at *Oregon Graduate Institue*, where Professor John Launchbury and the *PacSoft* group earned my immense gratitude by hosting me for four months. The change of environment allowed me to focus on the project work again, and during my stay *Visanti* was founded, and I was employed in a position equivalent to the one I had in *MindPass*. The thesis is almost entirely built on the work I have done for *Visanti*.

1.5 Overview of the thesis

This thesis is divided into three parts and a conclusion. The first part is about domain-specific languages, more specifically about developing software by designing and implementing a new domain-specific language. The second part concerns partial evaluation, which was my second major point of interest in this Ph.D. work. The third and last part describes a concrete case where a domain-specific language was invented to enhance a specific software development process. All chapters assume some familiarity with programming and software development.

Part I: Domain-specific language theory The first part of the thesis consists of two chapters. Both of these chapters relate to the situation where a software development challenge is met by introducing a new domain-specific language. The first section of Chapter 2 discusses why this approach may be a good idea (and why it may not). The main part of the chapter is a survey of methods for implementing such a language. Implementation of programming languages is notorically hard and there are many suggested methods, of which I review a selected subset. One of the methods elaborates on the connection between partial evaluation and domain-specific languages that we described above. On top of the survey, the reader is provided with some hints and warnings regarding each method's practical applicability in a given situation.

Chapter 3 relates to a much later stage in a software development project: the project evaluation. In the chapter, I argue that in a project where it was chosen to introduce a new domain-specific language, that choice ought to be evaluated. The chapter illustrates how other authors have described such evaluations in the literature. It also describes how the formulation of initial expectations regarding the domain-specific language may and should effect the choice of evaluation method.

The two chapters are both addressed to the practitioner, e.g. a leader of a software development project in which invention of a domain-specific language is considered a possible choice of action. The chapters do not require that the reader has a background in any specific academic areas, but familiarity with Bratman diagrams (also known as T-diagrams) makes Chapter 2 easier to read. I have written the text in a more direct style than the rest of the thesis, and in Chapter 2 the author steps forward in the singular “I”, rather than the plural form I chose for the rest of the thesis.

Part I can be read independently of the following chapters. Chapters 2 and 3 can be read independently of each other, although Section 2.1 (on the blessings and curses of domain-specific languages) provide useful background knowledge for reading Chapter 3.

Part II: Partial evaluation theory The second part of the thesis is divided into three chapters. The first two of these (Chapter 4 and 5) both report on an effort towards constructing a partial evaluator for the concurrent programming language *Erlang*. The instigator of this work was John Launchbury, and while the work ended up going in a somewhat different direction than what we talked about in Portland, my basic understanding of the problems were developed in discussions with John. The first chapter introduces the reader to the language and devises a formal semantics for a subset of *Erlang*. Such a semantics is a big help when discussing the many technical details of a language, as is necessary when defining a partial evaluator. It is also an absolute prerequisite for proving a partial evaluator correct, as has been done for a number of such systems.

Chapter 5 discusses partial evaluation of *Erlang*. I set out to write a complete partial evaluator for a reasonable subset of the language, but this goal turned out to be much too ambitious within the bounds of this project. The chapter demonstrates a number of example *Erlang* programs that could be partially evaluated with benefit. These examples motivate a reasonable modest definition of *which* program transformations a partial evaluator should be able to do. Such a definition is not obvious for a concurrent language. The chapter also reviews papers on partial evaluation of other concurrent languages and paradigms.

Chapter 6 is joint work with Robert Glück. The chapter has, in form of a regular paper, been accepted for publication by the *ACM Transactions on Programming Languages and Systems* (TOPLAS). The chapter discusses certain differences and lacks thereof between the two ways of performing partial evalution: the *online* and the *offline* approach. We prove that – in contrast to a common belief – for many languages, the two approaches achieve the same formal accuracy. This results leads to a more sophisticated view on the differences between online and offline partial evaluation.

A background in formal semantics, in particular sound knowledge of structured operational semantics, is a necessary prerequisite for reading all three chapters in this part of the thesis. Furthermore, Chapter 6 can hardly be understood without a firm background in partial evaluation. Chapter 5 is better appreciated with some understanding of partial evaluation too.

The second part of the thesis can be read independently of the two other parts. The *Erlang* chapters are also independent of Chapter 6, but Chapter 5 depends heavily on Chapter 4.

My partial evaluation theory is not applied in the last part of the thesis. This is a consequence of circumstances. During the Ph.D. project, I worked with several application ideas that involved partial evaluation, in particular of *Erlang*. But I had to consider my duties towards *Visanti*: the application that I ended up spending most of my time on, was the one that had the most promising potential for *Visanti* as a business. Maximizing the benefit of my employer took priority to knitting the different parts of this thesis more closely together.

Part III: MORI – an application of domain-specific languages The final part of the thesis presents a practical application of domain-specific languages that I carried out for *Visanti*. The application concerns the integration of a *Visanti* software product into any given network of servers and workstations that the customer might have. I designed a language for specifying some very important aspects of the product's customization, and wrote a compiler for the language. Chapter 7 describes the case and the project in much more detail. Chapter 8 discusses the design of the language and its compiler in more techincal terms. Finally, Chapter 9 is devoted to specifically to the internal optimization phase of the compiler. This particular detail deserves an entire chapter because of its relevance to the semantics of database query languages. In the chapter, I formally prove the correctness of certain powerful optimizations that are relevant to the execution of *SQL* database queries.

Part III of the thesis builds on the first part, and I recommend reading Chapter 2 and 3 before Chapters 7 and 8. There is a linear progression in Part III: Chapter 7 is a prerequisite for understanding Chapter 8, and both of these chapters are prerequisites to Chapter 9. The latter also assumes that the reader understands denotational semantics.

Conclusion The thesis ends with a conclusion in Chapter 10. The conclusion sums up the contributions of this thesis and points to directions for future work.

1.6 Acknowledgements

Many people people have supported my work directly or indirectly during the last three years. I would like to thank Neil Jones for all the things he has taught me over the years, and for his support during that ghastly period when the project seemed all but doomed. Robert Glück taught me pretty much everything I know on partial evaluation, and I want to thank him for that, and for being an absolutely splendid work companion. As I already mentioned, it's hard to express my gratitude towards John Launchbury and *everyone* at OGI for taking so good care of me, and for giving me many new perspectives on our favourite topics. Very special thanks goes to Emir Pasalic, Iavor Diatchki and Levent Erkök for making my stay in Portland so much more fun!

Coming back to Denmark, I must thank *Visanti* for their support, and all of my colleagues for being great through good times and bad times. The flexibility of the Danish Academy of Technical Sciences and Christian S. Jensen was essential to the completion of this project.

I thank my friends in the TOPPS group for making the past many years interesting and enjoyable. I must emphasize Jakob Grue Simonsen, who has been my daily source of entertainment and good advice during the last, intense year. Jakob was the prime proofreader on this thesis[1], and I also thank Peter Lund and Thomas Hildebrandt for helping me in that respect.

Last but not least, I thank all of my friends, my family and dear Line, for all those joys you give me.

[1]But don't blame him for my mistakes!

Part I

Domain-specific language theory

2 - Implementation of domain-specific languages

[...] two things were immediately clear:
(a) whatever they were doing was
enormously complicated and expensive, and
(b) they knew exactly how to do it.
Austin Grossman[1]

The main part of this chapter surveys a number of methods for implementing domain-specific languages. The aim is to give a practitioner an idea of the span of ideas that have been put forward in the programming language community, and to illuminate practical strengths and weaknesses of each method.

Before explaining *how* to implement a new DSL, Section 2.1 discusses the associated question of *why* – and *why not*! This section explains in detail the most important benefits and risks of introducing a new DSL. The rest of the chapter is really addressed to the project team that has already decided to go forward with the DSL approach but needs to find out *how* to progress. Section 2.2 gives the larger perspective on this question, while the following sections review literature on the different methods in more detail. Section 2.10 concludes.

2.1 Expectations to DSL introduction

In this section I will try to sum up the main kinds of benefits and risks that have been expressed about the introduction of a new DSL. I'll look at advantages and disadvantages separately. The main source of the suggestions is the bibliography by van Deursen, Klint and Visser ([vDKV00]).

Potential disadvantages Papers on DSLs of course report on projects where the invention of a DSL seemed like a good idea. Thus, it is not surprising that the authors anticipate fewer downsides than upsides regarding the introduction of DSLs. Nonetheless, the below issues can be very serious.

- *Complexity of language design and implementation.* This is certainly *the* main reason to avoid making a DSL. A DSL compiler/interpreter represents

[1]Recalling a visit to Universal Studios, and contrasting the movie business to the computer game business. In [Gro03], Introduction.

a *meta-solution* able to express every application you might want to write in the specific domain, and as discussed in the introduction (Section 1.2) such a meta-solution may be hard to find or even non-existent. A compiler or interpreter must also handle language-related problems (e.g. parsing, type checking etc.). This "extra layer" naturally means more work in design and implementation to begin with, even though writing each application may be easier afterwards. In many cases, the initial effort may not be worthwhile.

- *Harder maintenance.* Just as a compiler or interpreter may be harder to write than a number of applications in the first place, programmers may also find these types of programs more difficult to maintain. There may also be maintenance advantages to implementing a DSL, though. I'll discuss the potential advantages below.
- *Cost of education for DSL users.* While a DSL are most often designed to appeal to its intended users, there is always a learning effort associated with using a new programming language.
- *Slower applications.* Although some DSL compilers incorporate clever optimizations that makes the object code outperform hand-written solutions, things may also work out differently. Maybe the complexity of writing a compiler is perceived to be too large and you settle for an interpreter thus introducing an interpretive overhead. Or maybe a compiler *is* written but it turns out to be too hard to write a good optimizer. In some situations such a loss of efficiency is not acceptable.

Potential advantages Papers on DSL inventions tend to be written by researchers in programming languages who are mostly proponents of DSLs to begin with. As can be expected the literature points out many different potential benefits of DSL creation.

- *Productivity Increase.* This is often the main driving factor behind DSL introduction: the new language may allow programmers to develop their applications faster. One specific aspect is increased reuse of code; the code incorporated in the compiler/interpreter for the DSL will automatically be reused every time a program is compiled/interpreted.
- *Easier maintainence.* Changing back-end architecture may be hard when hundreds of applications have been written for the existing architecture. But if the applications were written in a DSL, only the compiler/interpreter needs to be modified to obtain a full conversion. Of course this is also means

increased portability, because all applications may be moved to another platform by just reimplementing the compiler/interpreter. Maintenance may also be easier if the DSL has been designed to make programs largely self-documenting.

- *Expanding the group of potential programmers.* By having a syntax that is understandable to non-programmers, a DSL may allow domain experts to program applications themselves. This may boost productivity directly by removing the need for programming assistance to these experts. By removing the “middle man”, it also reduces the risk of the code not corresponding to the intentions of the experts.

- *Greater reliability in applications.* A DSL compiler may effectively prevent the construction of incorrect or unsafe object programs. A DSL may also be designed to support the validation of desirable properties at the domain level, which is usually a lot simpler than verifying code in a general-purpose programming language.

- *Faster applications.* Some DSL compilers are highly optimizing, exploiting the fact that many more oportunities for optimization may be found in domain-specific code compared to code in a general-purpose language. Applications generated by such compilers may outperform all hand-written solutions in practice.

- *Conservation and explicitation of domain knowledge.* Some authors see in a DSL a possiblity for “wrapping up” organizational knowledge about a given domain, conserving it for future use when the experts may not be in the organization anymore. This scenario seems questionable, though – experience shows that *all* software must eventually (and sooner rather than later) be modified or updated to maintain its usefulness. For a DSL this will be hard without a domain expert at hand.

  On the other hand, the wrapping up may itself be useful, not as conservation of knowledge but as *explicitation* of it. A DSL may express in a clear way the spectrum of domain problems the organization has readily coded solutions for (has 0th order ignorance of, in the classification of [Arm00]). That is, by looking at the language – whether through a syntax or through representative DSL applications – one gets an overview of the organization’s knowledge within the specific domain. This overview may form a valuable basis for discussing development of the organizational knowledge.

Synthesis: dimensions of DSL impact If you look at the lists above, it should be clear that some of the advantages and disadvantages really belong to the same

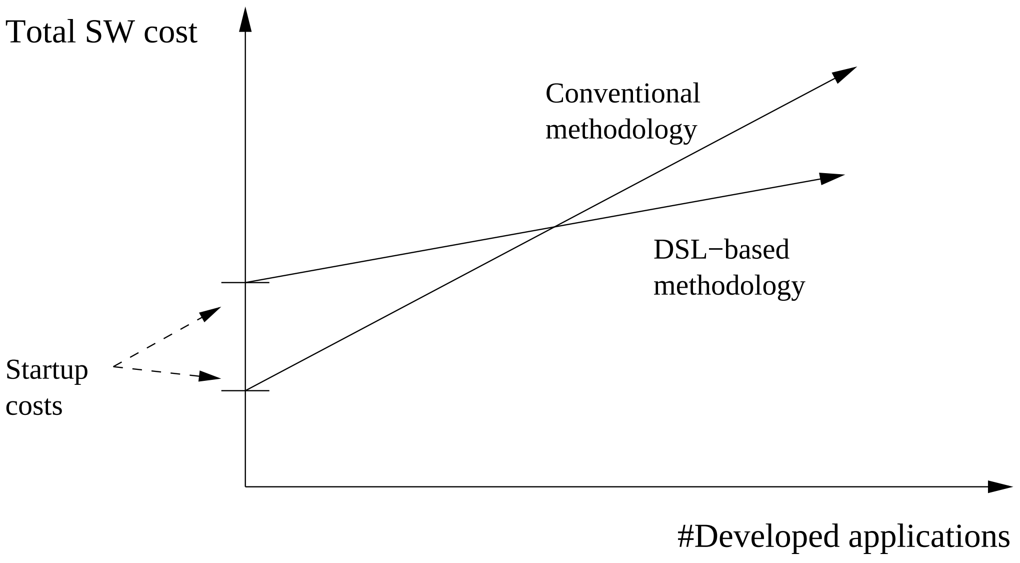


Figure 2.1: The payoff of DSL methodology. This figure has been adapted from [Hud98b].

dimension. To begin with, the main negative effect of *complexity of language design and implementation* is low total productivity. This is illustrated by Figure 2.1. The figure shows the total cost of software production – the resources spent on developing a number of applications – as a function of the number of applications in total. The idea is simple: the DSL-based methodology carries greater start-up costs (due to the complexity of DSL design and implementation) but usually allows applications to be developed more cheaply afterwards. So the total productivity is better with a DSL only if start-up costs, savings per application and the number of applications needed are right.

This boils down to one *productivity* dimension, in which the introduction of a DSL may prove to be an advantage or a disadvantage. It should also be clear that *maintenance* and *application efficiency* form such dimensions. The rest of the advantages/disadvantages listed above seem to be more one-sided.

The above reasoning yields the set of dimensions that is given in Figure 2.2. In some dimensions (like productivity) DSL methodology can be a blessing *or* a curse. In other dimensions you either expect DSL creation to be an advantage or to be a disadvantage.

Dimension	Potential disadvantage	Potential advantage
Productivity	$\surd$	$\surd$
Maintenance	$\surd$	$\surd$
Application efficiency	$\surd$	$\surd$
User education	$\surd$	
More potential programmers		$\surd$
Application reliability		$\surd$
Knowledge management		$\surd$

Figure 2.2: Dimensions of a DSL's impact on application development.

2.2 Bird's eye view of DSL implementation

"The arguments in favour of creating a DSL convince me," says the project leader. "We need a DSL. Our experts did a domain analysis, so now we have a general idea of what should be in the language. How do we implement it, so that we can execute our DSL applications?"

"You could write a library in the programming language that you ordinarily use", replies the author. "The concepts that your domain experts have listed could become class definitions (or data types, if you're into functional non-object-oriented programming), and the operations could become methods."

The project leader looks disappointed. "But the intention was that our domain experts (who are not programmers) should be able to write the DSL applications. They can't use a C++ library," he objects. "And I can't see how we can have verification and domain-specific optimizations on top of such a library."

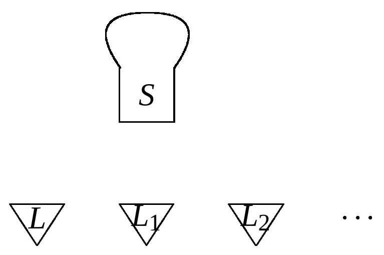
The author nods. "All right then, take this book ([ASU86]). It tells you all you need to know about parsing, intermedate code and code generation. Many compilers have been built using those techniques."

Flipping through the pages, the project leader complains: "That's a mighty book, and it sounds like hard work. Maybe the benefits from implementing a DSL that way will not outweigh the costs. Isn't there an easier way that still goes beyond writing a library?"

"Sure", I reply, "I'll give you a survey. But first we'll take a step back and analyze the situation a bit".

2.2.1 An illustration of the task

Consider a given platform supporting the execution of a number of languages, call them L, L_1, L_2, L_3 and so on. You want to execute a program (actually many programs) in your new DSL, call it S. In Bratman notation (see [Bra61, ES70]) the situaton can be illustrated like this:



ignoring that most of the *L*s are likely interpreted or compiled. The task is to connect the upper figure with one or more of the triangles in order to obtain an execution of the DSL program.

2.2.2 Practical issues in DSL implementation

When choosing how to solve this task, one should be aware of the inherent practical consequences of each method. These will not always be apparent to non-experts from reading just one paper. The potential issues that I would like to point out are the following:

User-oriented issues Who are the intended users of the DSL, i.e. the application programmers? Depending on their background and programming skills it may be required that the DSL syntax fits certain limitations; caters to their way of viewing things. Thus, if a method itself enforces limitations on DSL syntax, it may not be a good choice.

Domain-oriented issues For some domains, certain very specific functionalitites must be in the implementation language, for example a *CORBA* library or floating number computations that follow IEEE specifications strictly. If a method builds upon one specific implementation language (and in particular if this language has a comparatively small user group and thus probably also a small set of available libraries), it may not be be a good choice.

Integration Applications have to work in specific system contexts. If DSL applications must integrate smoothly in a large C++ project, they probably should be implemented in C++. Also, if a DSL application must run on your customer's platform, you might not trust third-party software being glued into it. And if the DSL applications must run on a very specific platform, say, if they are embedded applications, the implementation languages must be available on that platform.

Team expertise Some methods require expertises outside the application domain and popular programming languages. This may also be a challenge for a given project team.

There are of course other many other issues I could have looked at. For instance, certain projects may have very specific requirements to application efficiency, portability, reliability or documentation. Some methods may not be suited to fulfill such requirements.

The issues that *were* listed above can be stated as questions to a given method for implementing DSLs:

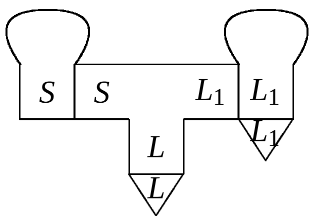
1. Does the method put limits to the syntax of the domain-specific language *S*?
2. Does the method limit the choice of implementation language(s) $L, L_1, \ldots$, and if so, which kinds of languages are you left free to choose between?
3. Does the method imply the need for using special tools before or during the execution of DSL applications?
4. Does the method require special expertises of the DSL-implementing team?

The Bratman notation will be used in the following to help illustrate each method's relation to the first three questions.

2.2.3 The methods in brief

Embedded DSLs The first suggestion – implementing the DSL as a library – accomplishes the implementation task by simply demanding that the target language *S* can be embedded in an existing language *L*. Of course, this method puts severe limits on the syntax of *S*: it must abide the syntax of *L*. Some papers argue that this can be blessing in itself, and they also demonstrate that by choosing *L* right, the solution is more flexible than you would expect. Note that the "right" choice of *L* often means a programming language that is not among the most widespread ones. Section 2.3 reviews papers on embedded DSLs. Some of these papers also discuss implementation of DSL interpreters. This approach is not often addressed directly in the literature, and there is no separate section about interpreter techniques in this chapter.

DSL compiler writing techniques The second suggestion that the project leader was given above, was to build a compiler, in Bratman's notation:

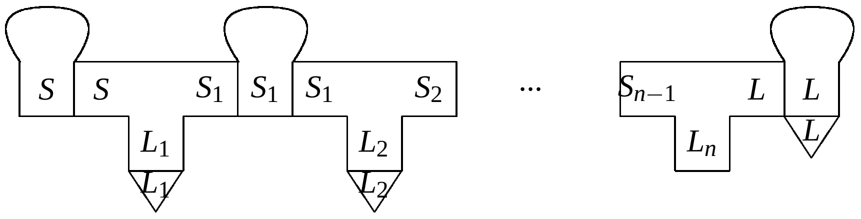


As mentioned, writing a compiler is not usually an easy task.[2] Several papers suggest compiler implementation techniques relevant for DSL compilers. Some authors advocate implementation methods that use facilities of particular languages (choices of L). These methods expect S to be an extension of L, but leave the choice of L_1 free.

As discussed in Chapter 1, partial evaluation can also be used for achieving compilation. Writing a compiler using partial evaluation offers freedom in the choice of S syntax, but puts some limits on the choice of $L = L_1$: that language must have an associated partial evaluator. Furthermore, the implementation team must have some understanding of partial evaluation. A technique with similar effect is the use of staged interpreters. This technique puts severe limits on the choice of L.

The papers on compiler writing techniques are reviewed in Section 2.4.

Lightweight compiling One approach to compiler writing is a bit different from the above. One paper suggest implementing a compiler *in small steps*, each of which transforms the application a bit towards the final target language (see Section 2.5):



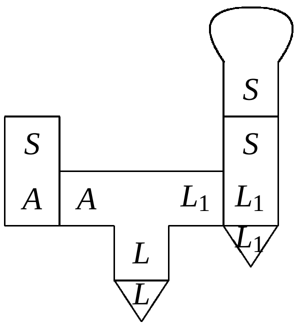
One of the motivations is that each step should be much easier to write than a full-scale compiler. The paper suggests that the Ls be small languages rather than general purpose programming languages, and that the syntax and features of S be severely restricted. One should be aware of the risks associated with fragmenting the DSL implementation into many code pieces, and with the dependency on many implementation languages rather than just one.

[2]Which is one good reason most computer science students have to do it.

Extendible language implementations Others again provide an interpreter or a compiler specifically built for extension. This means the DSL developers will not have to build one from scratch, and instead of having to know programming language theory, they will only have to understand the interpreter or compiler they are extending. Clearly, the choice of both source language S and implementation language L will be limited to choices for which an extendible implementation exists. The extension appoach is covered in Section 2.6.

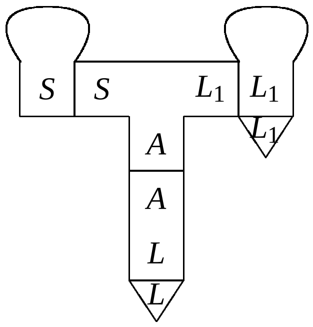
DSL interpreter tools The remaining categories concern the use of tools to aid DSL implementation. The difference between technique and tool is very blurry, and I have somewhat arbitrarily put e.g. partial evaluation and extendible compilers in the above technique sections.

The first category of tools provides the DSL implementor with a language (i.e. *another* domain-specific language) A for writing interpreters in:



Such tools provides the compiler from the language A to the final implementation language L_1. A paper describing this kind of tool is reviewed in Section 2.7. The tool only supports one choice of L ($= L_1$), but does not constrain the design of the source language S. And of course, one will have to learn the A language to use the tool.

DSL compiler tools A large number of people have developed tools for writing DSL compilers. Most tools provide a language for writing compilers:



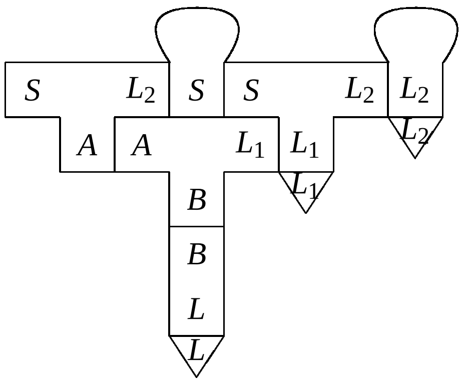
Here, the A-interpreter is part of the tool. These tools do not generally restrict the choice of S and L_1, but L is of course determined by the given tool. And again, one will have to learn the A language to use the tool.

Several papers describing this kind of tools are reviewed in Section 2.8.

Two-step compiler writing One tool does not fall into the two categories above. This tool lets you implement DSLs in two steps:

- First you specify a language A for writing compilers in. This is done by writing a compiler from A to an executable language (L_1) in the tool's own language B.
- Then you write a compiler for the DSL in your new compiler language A.

The complete situation looks like this:



The languages S, L_1 and L_2 can be chosen at will, but L is fixed and B must be learned. The choice of language A is very limited: programming in A is done by choosing between a (potentially large) number of options through the tool's graphical user interface. The tool is described in Section 2.9.

2.3 Domain-specific embedded langauges

Although the idea of implementing a DSL by adding domain-specific features to an existing programming language may seem like an immediate choice, and indeed can be traced back to the "next 700" paper by Landin ([Lan66]), most papers on this technique suggest its use as a counter-move to avoid problems with other approaches.

Lambda: the ultimate "little language" One such paper is [Shi96] by Olin Shivers. The author lists a number of existing DSLs: Unix tools for scripting, programming and more, and cites some of the advantages of DSLs that were dicussed above. He then moves on to discuss disadvantages of DSLs, treating in particular detail two issues (related to the *User education* dimension in Section 2.1):

- That many DSLs are badly designed – and all different! – when it comes to features that aren't domain-specific like lexical conventions and general programming features (e.g. loops, conditionals, variables).

- That coordination between different little languages is usually very primitive, e.g. applications can only interact through Unix pipes.

The suggested solution is to embed DSLs into "a powerful, syntactically extensible programming language, such as Scheme". This allows the DSL designer to focus on domain-specific features alone, while lexical conventions and other more general features come for free. It also means that the host language can be used to glue applications together in a tight way.

Shivers provides two examples of languages embedded in *Scheme*: a language for controlling Unix processes and directing their inputs and outputs, and a *awk*-resembling language for writing programs that process text files based on pattern matching. Both languages are implemented by writing appropriate *Scheme* libraries and macros.

The author concludes that benefits of the embedding approach to DSL implementation are that it is eaier than inventing whole new language (citing a much smaller code base of the *Scheme awk* implementation than the original implementation), that the quality of the little language produced is greater (because it lends well-designed features from the host language), and that languages compose better (because complex data structured can be passed and shared). He notes that the unusually sophisticated macro system of *Scheme* is a crucial mechanism for properly embedding a DSL.

Other authors clearly disagree on this point. Hudak, in e.g. [Hud98b] and [Hud98a], lists many illustrative examples of DSLs embedded in *Haskell*. Hudak demonstrates how *Haskell*'s remarkably flexible syntax can be utilized to embed many different DSLs in the language. He lists higher-order functions, lazy evaluation, polymorphism and type classes as crucial for "pure embedding". Hudak also discusses how monads ([Wad95]) can be used to write interpreters for DSLs in *Haskell*, and he discusses how interpretative overhead may be removed through partial evaluation (see next section).

Yet other languages have been proposed for DSL embeddings. In ([FGS97]), Fromherz et al. argue that *constraint programming* is "an appropriate generic framework for domain-specific languages".

2.4 Techniques for writing DSL compilers

Compiling Embedded Languages The embedding approach discussed above can also be extended to be a compiler writing technique. This technique was pioneered by Samuel Kamin in [Kam] and [Kam98]. An extension of his work is presented by Elliott, Finne and de Moor in [EFdM00].

The three authors present their implementation of the language *Pan* for describing image synthesis and manipulation. They first illustrate how one would ordinarily embed *Pan* in *Haskell*, defining e.g.

```
type Image     = Point -> Color
type Point     = (Float,Float)
type Color     = (Float,Float,Float,Float)
type Transform = Point -> Point

translate (dx,dy) = lambda(x,y) -> (x+dx,y+dy)
scale (sx,sy)     = lambda(x,y) -> (sx*x,sy*y)
```

where `lambda` represents λ abstraction.

This embedding is then changed into a compiler by replacing values (like floats and float tuples) with program fragments. For example,

```
data FloatE = LitFloat Float
            | AddF FloatE FloatE | MulF FloatE FloatE | ...
```

Host language features such as tupling can still be used:

```
type ImageE     = PointE -> ColorE
type PointE     = (FloatE,FloatE)
type ColorE     = (FloatE,FloatE,FloatE,FloatE)
type TransformE = PointE -> PointE
```

The authors show how these algebraic data types can be used for implementing both optimizations and code generation.

As mentioned above, Kamin presented similar DSL compiler implementations in [Kam] and [Kam98]. His compilers were embedded in *ML* and represented program fragments as strings rather than algebraic data types. This representation does not allow easy implementation of optimizations. A paper by Leijen and Meijer ([LM99]) shows how the authors implemented an optimizing compiler from a database query language into SQL by embedding it in *Haskell*. And in their book [CE00], Czarnecki and Eisenecker demonstrate many techniques for writing program generators, of which DSL compilers are a special case, in *C++*.

Partial evaluation In the introduction (Section 1.3), I sketched an example of using partial evaluation to remove the efficiency overhead of interpreting a DSL. The effect of specializing a DSL interpreter to a given DSL program is equivalent to compiling the program. Partial evaluation can even take this a step further and rewrite a given DSL interpreter into a compiler. This is usually a help to the compiler writer as interpreters are generally easier to write than compilers. The techniques behind this are explained in detail by Jones, Gomard and Sestoft ([JGS93]).

A suggestion of how to apply partial evaluation in DSL development is given by Consel and Marlet in [CM98]. The paper outlines a DSL development method which is presented as a sequence of steps, but the authors stress that in practice the process needs to be iterated. The steps are as follows:

1. Perform a thorough analysis of your requirements, the technical literature and existing software base. Methods may include commonality analysis ([Wei]) and domain analysis ([CE00]).

2. Refine the above analysis by formulating "major concepts" as types and operations on these (in form of a *semantics algebra*), and by formulating a syntax and informal semantics for your DSL.

3. *Stage* the informal semantics by figuring out what information will be available at compile-time (i.e. when the DSL program has been written but before it is running), and deciding which of the semantical operations can and should be performed at compile-time rather than run-time.

4. Write a formal, denotational semantics for your DSL. In other words, write a compositional mapping from DSL programs to compositions of operations in the semantic algebra.

5. Reformulate the semantics to map to an *abstract machine* (see the paper for details).

6. Implement the abstract machine and the semantics from the previous step. These form an interpreter for the DSL.

7. Now apply partial evaluation to obtain compilation rather than interpretation, improving the efficiency of the implementation.

This rather long sequence of steps serves as a recipe for developing well-designed DSL compilers. The paper elaborates on the benefits of approaching compiler development through the above process.

DSL Implementation Using Staging and Monads Sheard, Benaissa and Pasalic describe a related approach to DSL compiler development in [SeaBP99]. A major difference is that instead of partially evaluating an interpreter in a standard programming language, the authors recommend writing the interpreter in a *staged* language like *MetaML* ([TBS98]).

The paper focuses on the implementation phase and divides its suggested approach into three steps:

1. Write a basic, functional-style, compositional interpreter. To handle effects, make it a *monadic interpreter* (see [Wad95] for an explanation of this programming style).

2. Now rewrite the interpreter into a *staged interpreter*. This is essential and needs some explanation. In a staged program, every expression (or command, if the language is imperative) has an associated *stage* which is a non-negative integer. Running a staged program effectively decreases the stage of expressions by one. Expressions of level zero are evaluated to ground values, possibly containing "encoded" positive-level subexpressions. There are of course dependencies: if the evaluation of an expression on stage n depends on the value of another expression, this other expression must be on a stage smaller than n. This is explained in e.g. [TBS98].

   A staged interpreter typically has two stages: one for compile-time and one for run-time. In a well-designed staged interpreter, any choice that depends only on the source program text is encoded as an expression on the compile-time stage. An expression whose evaluation depends on run-time input must of course be on the run-time stage.

   Running a staged *MetaML* interpreter on a DSL source program will yield a compilation of that program into one-stage *MetaML*, i.e. into *ML*.

3. Add postprocessing of the code generated by the staged interpreter. This may be optimization or code generation, if the final target language is not *ML*.

The overall effect is very close to the one obtained by using partial evaluation to develop a DSL compiler: you get the simplicity of writing an interpreter together with the efficiency of compiled applications. With the staged approach the price is that you must write the interpreter in a staged language (of which there are preciously few) and you have to separate compile-time and run-time stages manually. With the partial evaluation approach, the choice of language is wider, as partial evaluators have been developed for e.g. *C*, *Java*, *ML*, *Scheme* and *Fortran*. In principle you do not need to separate the stages in your interpreter as this is done by the partial evaluator itself (often by a so-called *binding-time analysis*, see [JGS93] and Chapter 6). In practice the process is interactive, as one must write the code in a style that allows the partial evaluator to discover the right staging.

2.5 Lightweight Languages as Software Engineering Tools

A different approach is suggested by Spinellis and Gurprasad in [SG97a]. The paper reviews a number of different software applications written by the authors. The development of each application involved compilation of one or more *lightweight languages*. A lightweight language refers to a domain-specific language with quite simple syntax and semantics (this is of course a subjective concept).

The authors sum up their experiences from these projects in a list of implementation techniques – in the form of advices or suggestions – that they have succesfully used on their lightweight languages. Two main pieces of advice are: to use existing tools, and to use these in any combination you see fit. Tools are typically interpreters, either of very simple macro languages (*awk*, *cfront*) or of more powerful scripting languages (*perl*, *ksh*). Even *lex* and *yacc* can be considered interpreters of relatively small languages. The point is that a number of small transformations implemented with these tools can accomplish compilation of many useful, small languages.

Regarding the design of these small, lightweight DSLs, Spinellis and Gurprasad advocate that features from the “tool languages” (e.g. the ones named above) and from the target language be used in your DSL whenever needed. For example, if the target language is *C* you may allow programs in a lightweight DSL to contain `#line` declarations that are passed directly to the compiled code and thus let the programmer find errors (generated by the *C*) more easily. Another example is to allow *perl* code in a DSL that is processed by *perl*. The reflexive function *eval()* in *perl* lets us implement that feature essentially for free.

The authors also suggest that one adapts DSL syntax and lexical matters to the tools rather than the other way around. An example is choosing a line-based syntax, which will allow many Unix tools like *awk* much easier to use in the implementation.

The authors note some potential problems with using their approach. They especially stress the fact that maintaining a system based on many languages can be a serious problem for the system developers.

2.6 Extendible compilers and interpreters

If writing a compiler or interpreter from scratch is too much of a mouthful, maybe you can just extend an existing one? While rewriting other people's code is usually a risky approach to software development, some compilers and interpreters have been built explicitly to support this kind of work. The *Tcl* interpreter ([Ous94, Ous98]) is a well-known example. Engler et al. ([EHK96]) wrote an extendible *ANSI C* compiler, and so did Stichnoth and Gross ([SG97b]).

Jargons An even more ambitious effort is the *Jargons* project as described by Nakatani and others in [NJ97] and [NAOP00]. The goal of this project is not only to support the implementaion of DSLs but also to integrate these seamlessly. Jargons are in intention an XML ([HM02]) for programming.

A jargon is a DSL which is based on one specific abstract syntax (called *WizTalk*) and which is interpreted by the *InfoWiz* generic interpreter. The syntax allows the application programmer to build trees with a keyword in every node and associated keywords, strings and such at leafs and internal nodes. A *WizTalk* document (i.e. a program) may use several jargons at a time. Indeed, the authors promote a style in which any feature that could be useful on its own is made into a single jargon. They argue that the common syntax prevents the emergence of a Tower of Babel.

To interpret a program, a domain-specific plug-in is loaded into the *InfoWiz* interpreter for each jargon used. The interpreter coordinates namespaces between the different jargons loaded and figures out for each keyword which plugged-in function to call on the given node.

The interpreter is itself written in a language called *Fit*, but the authors note that another "language with garbage collection, good facilities for text processing, and a flexible function invocation environment could have been used instead" ([NJ97],p. 62).

2.7 A tool for writing DSL interpreters

Integrating Domain Specific Language Design in the Software Life Cycle Kutter, Schweizer and Thiele describe the *Montages* approach to DSL implementation in [KST98]. Their *Gem-Mex* system provides an integrated environment for writing DSL interpreters, with a graphical editor, debugger and more.

A Montage is in essence a self-contained EBNF production with associated semantics. The semantics is described using Gurevich's *Abstract State Machines* (ASM) – a semantic formalism in which a source program is mapped to an abstract imperative program performing updates on a state. The state of an ASM is a model of a first-order logic with a finite number of predicates and function symbols.

The *Mex* (Montages executable generator) part of *Gem-Mex* is a program that compiles a set of Montages into an interpreter of the language they describe. Both *Mex* and the resulting interpreter are written in *C*. The system also produces documentation in *LaTeX* and *HTML*. The authors conclude that the modularity of Montages along with the auto-generated documentation makes languages developed in *Gem-Mex* essentially self-documenting. They also point out advantages of the modularity, extendibility and portability of the approach, and they argue that the efficiency of interpreters generated by the system have been quite satisfactory to their needs.

2.8 Tools for writing compilers

There are many tools in this category, including *lex* and *yacc* of course. I'll focus on tools that are specifically made for writing DSL compilers. Some other tools, like Østerbye's *Refill* ([CØV02]), aim at providing extensions of existing languages (like *Java*), and could thus be used for embedding a DSL in e.g. *Java*.

ASF+DSF van Deursen, Heering, Klint and others have worked with the combination of *Algebraic Specification Formalism* (ASF) and *Syntax Definition Formalism* (see e.g. [DHK96, BDK$^+$96]). This combination, and their *Meta-Environment* tool supporting it, implements DSL compilers as transformations on abstract syntax trees (ASTs).

The grammar of a language L to be implemented in ASF+DSF is defined by a set of functions for constructing its ASTs. This definition is roughly equivalent to writing an ordinary BNF grammar for L. You can also specify

- Transformations on the source language as functions from L-ASTs to L-ASTs.
- Type checking as functions from L-ASTs to boolean values.
- Compilation to a target langauge L' as functions mapping L-ASTs to L'-ASTs.

A definition may also contain equalities on ASTs (to specify e.g. that ASTs for sets $\{1,2,1\}$ and $\{1,2\}$ should be considered the same).

The *Meta-Environment* executes ASF+DSF specifications. Given a specification of a language L, the system will generate an L-parser, an L-rewriter (possibly transforming L-ASTs to target language L'-ASTs) and a *pretty-print generator* that generates target code. The *Meta-Environment* also documents *L* in a *LaTeX* document.

The CWI researchers have reported on several projects where the system has been successfully used to introduce DSLs in an industrial context ([BDK$^+$96]).

KHEPERA: A System for Rapid Implementation of Domain Specific Languages The paper [FNP97] by Faith, Nyland and Prins describes the *Khepera* system, a tool for writing DSL compilers, not unlike the ASF+DSF *Meta-Environment*. Just like the latter, compilation is done in a number of small steps, each one transforming an abstract syntax tree to another one. The transformations are written in a language specific to the Khepera tool. This language provide many mechanisms including all kinds of tree traversal schemes and complex tree matching.

On the grand scale the authors expose two *Khepera* features that distinguishes it from ASF+DSF. First of all, the DSL creator writes the parser for his DSL in *Lex/Yacc* rather than a proprietary format. Second, *Khepera*'s rewriting engine that performs the tree transformations keeps track of all rewritings in order to support very flexible debugging of DSL programs.

The metafront System: Extensible Parsing and Transformation Yet another tool that can be used for DSL compilation is *metafront*, the ways of which are explained by Brabrand, Schwartzbach and Vanggaard in [BSV03]. This system is also akin to ASF+DSF, but it differs notably in the kind of transformations it allows the programmer to specify. The authors stress that – unlike ASF+DSF – the *metafront* system

- Is guaranteed to terminate with a well-defined result when transforming a DSL program.
- Is both theoretically and practically very efficient.

Safety and efficiency are key points of *metafront*, and compared to ASF+DSF they are obtained by restricting the allowed transformations (ASF+DSF transformations are Turing complete).

Definitions in *metafront* take the form of annotated EBNF rules. A rule defines how a term matching the given production can be transformed into another term. The system employs a very sophisticated scheme called *specificity parsing* to determine which among a number of productions matching a given term should be used during transformation. It also employs sophisticated static checks to ensure that a language specification performs a well-defined and terminating transformation.

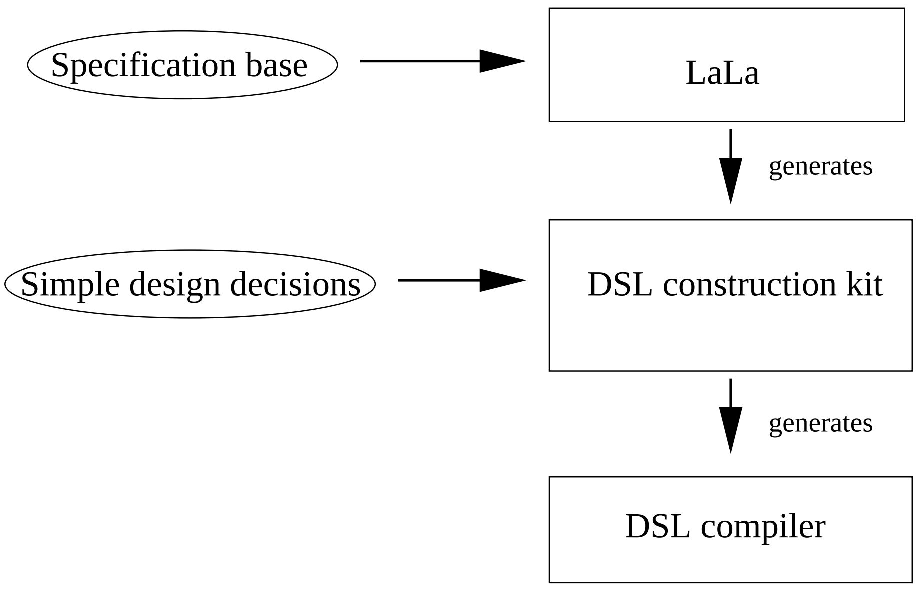


Figure 2.3: Implementing a DSL with *LaLa* in two steps.

2.9 Language Design and Implementation by Selection

In the paper [PK97], Pfahler and Kastens describe the *LaLa* (an abbrieviation of “Language Laboratory”) system for designing and implementing DSLs.

Implementing a language in the system is a two-step process called *design and implementation by selection*. The process is illustrated in Figure 2.3. In the first step, a *specification base* is given to the *LaLa* system which then generates a specialized DSL construction kit. This kit is a graphical tool that lets the language designer make a lot of simple decisions through appropriate clicking and filling in forms in a nice graphical user interface. When all relevant design choices have been made, the construction kit outputs a compiler for the specified DSL (making use of the *Eli* compiler construction tool).

The specification base describes a “language class”. The authors mention having written two of these: *MicroImperative* defining a set small Pascal-like imperative languages, and *RepGen* defining report generator languages on literature databases. The main component of these definitions is parametrized input to the *Eli* compiler generator. The parameters correspond to the decisions made in the DSL construction kit. For *MicroImperative* these parameters include:

- Whether semicolon is a separator or a terminator of statements.

- Whether scopes are Algol-like or C-like.
- Whether comments are Pascal-like or C-like.
- Whether variables must have explicit declarations.
- If not, the implicitly assumed type (integer, real or boolean) of undeclared variables.

As the two last items indicate, the parameters may have internal dependencies. The construction kit performs consistency checks on these dependencies and signals any inconsistencies to its user.

The philosophy behind the two-step process is to enable domain experts to design DSLs. Once a reasonable specification base has been written, domain experts (who may not be programmers) can click their way to a final language design. Programming language experts can thus concentrate more on writing the specification base and less on the domain-specific details.

2.10 Conclusion

The first section of this chapter listed the most important benefits and risks of introducing a new DSL. The different aspects were collected into *dimensions* of a DSLs impact on the work processes it fits into.

The benefits listed in Section 2.1 explained the *why* of DSL creation. The main part of the chapter explained the *how* by providing a survey of the litterature on methods for implementing DSLs. The survey covers more methods than any other DSL implementation text the author is aware of. But the size of the relevant body of literature prevents mentioning every paper and every tool. Instead, I have aimed at presenting a representative selection, with a bias towards those methods that seem to have received most attention.

One reason for making this survey was to dispell possible misconceptions. Many papers on implemeting DSLs mention only one method. Thus, the project leader who happens to read just one paper may be led to think – mistakingly – that there is just “one true way”. Another motivation has been to illustrate some of the *practical* strengths and weaknesses that seem inherent to some of the methods. While much longer texts could be written on the subject, the present chapter should be a useful guide in the hands of the DSL implementation teams it was addressed to.

3 - A note on the evaluation of DSL creation

The truth can't hurt you
it's just like the dark
It scares you witless
but in time you see things clear and stark
"I Want You", Elvis Costello

This chapter is aimed at the reader who just finished a software development project in which a new DSL was created. It is meant to be a brief discussion about methods for evaluating whether inventing a DSL was worth the trouble. We'll argue that no matter what your technological goal was, you could always achieve it *without* developing a DSL (Section 3.1). This observation implies that approaching your goal through DSL invention is an actual choice and not just a *fiat*. Drawing from DSL projects described in the literature (Section 3.2), we'll try to sum up which expectations authors have expressed regarding their DSLs and how they've evaluated whether these expectations turned out to hold (Section 3.3). The discussion in Section 3.4 points to strengths and weaknesses of each evaluation method. Section 3.5 concludes.

3.1 Introduction: the choice of DSL creation

So you chose to design and implement a DSL. There was of course an alternative. No matter whether your DSL was implemented by means of an interpreter, a compiler, as an extension of an existing language or by other means, each of your DSL programs correponds somehow to a program in one or more languages that existed prior to your DSL.

If the DSL was implemented by writing a compiler for it, it is obvious that – from a technical viewpoint – all results could have been obtained without the DSL. Each DSL application is compiled into another language; one that existed prior to the DSL. So all application could (at least theoretically) have been written in this other language to begin with.

If the DSL is interpreted the same reasoning applies. Each DSL application corresponds to a *specialization* of the interpreter. This specialization is a program in the same language, the interpreter is written in. Again, we might (theoretically) have written this specialization ourselves, avoiding the creation of the DSL.

So no matter the implementation style, we could have avoided code generation (or interpretation) completely. Of course, this reasoning also applies to ordinary programming languages: theoretically, all programs could be written in assembly language. History has proven beyond doubt the value of high-level programming languages over assembly, but the case may not be so obvious when comparing a new DSL to an ordinary programming language.

There are also a number of alternative *generative* approaches to programming (see [CE00] for a collection) but we will focus on the active choice of a DSL-oriented solution over a nongenerative one.

Unless you just happen to have a fetish for writing grammars, parsers and compilers, there should be some kind of rationale behind the choice. That is, when you made the decision of developing a new DSL, you very likely had expectations about the advantages and disadvantages of introducing a new language, and you must have anticipated that the advantages outweighed the disadvantages.

With the delivery of your DSL and its accompanying tools, the question naturally arises whether your expectations were right, and whether the choice of introducing the DSL was a good one. We should therefore look for evaluation methods that address such questions. The following section should provide help in this direction, as we sum up what other DSL inventors have done to evaluate *their* projects.

3.2 Evaluation methods

In this section we review some of the most well-documented examples of evaluation of DSL impact. We have divided the papers into four groups, depending on their general approach to the evaluation. We dub the four categories the *experimental* approach, the *formal* approach, the *organizational*[1] approach and the *engineering* approach.

Put very briefly, the experimental approach is to perform an evaluation by means of controlled, scientific experiments. The formal approach is to evaluate a DSL and its tools by investigating formally defined properties of these. The organizational approach to evaluation is based on studies of the organizations that use a given DSL and its tools. The engineering approach is to perform an evaluation by means of *demonstration*, showing how certain applications can be engineered using a DSL and its tools. The four approaches are presented in random order.

The experimental approach Sometimes it is possible to test scientifically whether initial hypotheses about DSL impact can be proven correct. In a project by Kieburtz

[1]For lack of a better word, really.

et al. ([KMB+96]), a DSL called MTV-G was invented. A program in MTV-G translates and validates an electronic message in a format that complies with the format specified in the so-called C^3I system. The pre-existing alternative was that the programmers could modify a number of Ada program templates. The explicit hypotheses were that in comparison to the template approach:

- MTV-G would match the existing system's flexibility.
- MTV-G would increase the productivity of the programmers.
- There would be fewer defects in programs written with MTV-G.
- Programmers would perceive MTV-G as easier to use.

These hypotheses were meticulously investigated through experimentation. Programmers were placed in a controlled environment and developed applications both with and without MTV-G. Much data was recorded and collected, including e.g. the time spent on developing each application until it passed a reliability test. After the experiments, the involved programmers were also interviewed. This data was analyzed to see whether the four intended avantages were realized in practice.

The formal approach Some authors have pursued a more formal kind of test. In contrast to the MTV-G project, the experiments in Réveillère et al ([RM01]) did not involve putting people in controlled environments and measuring their performance. The authors avoid such installments by specifying their hypotheses in a formal manner rather than in more broad terms like "programmer productivity" and "ease of use".

The domain-specific language *Devil* is used for specifying device drivers. The specifications are compiled into low-level C, which is also the ordinary (pre-*Devil*) programming language for these drivers. The objective is that drivers written in *Devil* should be more *robust* than drivers written directly in C. A system is said to be robust when programming errors are generally caught early in the development process.

Réveillère and his colleagues choose to focus on the detection of typographical errors and inattention errors. While typographical errors may be easily caught in actual C code, detecting these errors in e.g. hexadecimal constants – of which there may be many in device drivers – can be non-trivial.

The authors use a method called mutation analysis to estimate whether their robustness goal was met by the *Devil* system. First they define an error model which clarifies the meaning of "typographical errors" and "inattention errors". These definitions are then formulated as mutation rules that specify how one can introduce one such error in a program by through rewriting (mutating) it automatically. Rules are formulated for both C and *Devil*.

The automatic transformations are then applied to a specific driver that comes in both a C and a *Devil* version. This yields about 2000 mutants of each version, 25% of which are chosen at random. It can now be tested whether the static checks in the *Devil* system generally catch the introduced errors earlier in the development process than errors are caught in the traditional approach.

The organizational approach Few papers report on experiments or formal testing. Refraining from such activities does not mean one has to base evaluations on speculation alone. Some authors gather data in the way you often do in the social sciences: through observations on the use of the DSL after its launch. The RISLA language is used by a Dutch bank for defining interest rate products. The langauge was developed by researchers at CWI in Amsterdam. Reporting on the RISLA project, van Deursen and Klint write:

> At the positive side, the RISLA project has met its targets: the time it costs to introduce a new product is down from an estimated three months to two or three weeks. Moreover, financial engineers themselves can [...] compose new products. Last but not least, it has become much easier to validate the correctness of the software realization of the interest rate products.

Similar *post factum* observations have been used in papers by Mogensen (in [Mog02] and [Mog03]).

Interviews could also form the basis of such empirical, but nonexperimental evaluation. As mentioned above, interviews were performed in [KMB^{+}96].

The engineering approach Other DSL inventors investigate the usefulness of their DSL by demonstrating its use. A well-documented investigation of this sort is the work on GAL by Thibault and others.

GAL is a language for specifying video adapter device drivers. The goal of GAL "was to improve the development time of video device drivers" ([Thi98], p. 12). This goal was found to be accomplished in [Thi98] and [TMC99]. The evaluation took form of a reimplementation of a number of existing drivers. That is, the researchers chose a number of examples and specified these in GAL. The measure of success was the number of lines of code in the original C code vs. the number of lines in the corresponding GAL specification, taking also into account the number of lines in the GAL interpreter.

The same evaluation approach and measure of succes was used in [SG97a].

Another illustrative example of the engineering approach to evaluation was performed in Orwant's Ph.d. thesis ([Orw99], see also [Orw00]). His EGGG language for specifying games was demonstrated to be quite flexible and open to programmer experimentation, when in a series of six easy steps the specification of Tetris was morphed into a specification of Pong. Each of the five intermediate specifications defined a perfectly good, playable game.

3.3 DSL expectations revisited

An evaluation should somehow analyse whether the initial expectations of your DSL-based approach turned out to hold or not. Recall that we identified seven dimensions of such expectations in Section 2.1:

- Productivity
- Maintenance
- Application efficiency
- User education
- More potential programmers
- Application reliability
- Knowledge management

To be fair, an evaluation of a DSL-inventing project should be relative to *explicit* hypotheses about the project's impact; hypotheses that were formulated before or in the early phases of the project. We expect the hypotheses to be related to the dimensions listed above. But the dimensions say nothing about *the object of study* in a given hypothesis: what the hypothesis expresses an expectation *to*. When we review the examples in Section 3.2, we find three different such objects of study:

1. The DSL and its tools.
2. The software development process that the DSL and its tools are supposed to fit into.
3. The organization whose processes the DSL and its tools are supposed to fit into.

An example of a hypothesis that fits into the first category could be the expectation that the *Devil* system is able to catch more typographical errors than the alternative application development system. An example of the second kind of hypothesis is the expectation that MTV-G would increase the productivity of typical programmers. The implied expectation of van Deursen and Klint that RISLA would substantially decrease the product release time is an example of a hypothesis of the third category.

Clearly, the enumeration reflects a certain progression in the generality of the object of study. But the borders between our categories are not clear cut. For instance, it is not always clear whether the formulation “a programmer” refers to any person equipped with the relevant skills or specifically to the people that end up working with the DSL in the relevant organization.

The object of study of a hypothesis is more or less orthogonal to the dimensions it is related to. The afore-mentioned expectation – that the *Devil* system is able to catch more typographical errors than it alternative – could easily be formulated with address to the more general objects of study. An alternative definition of a typographical error could have been e.g. the recognizable typos that a properly instructed programmer with at least six months experience in device driver programming *does make* during an 8-hour work day. Had this definition been used instead of the formal one, we would put the resulting hypothesis in the second category. The hypothesis that e.g. “by using *Devil*, company X will need to release 25% fewer patches to their device drivers over a 1-year period than they have until now” would be of the third category.

Note that there is a certain sense of implication between the three hypotheses: the verification of the original *Devil* hypothesis gives evidence to the plausability of first alternative hypothesis. Likewise, an experiment supporting the first alternative hypothesis would give evidence to the plausability of second alternative hypothesis.

3.4 Discussion

There may be many reasons for choosing one object of study instead of the others. One advantage of the formal definition of typographical errors in [RM01] is that it leads to a hypothesis that can be established with absolute certainty. An even more interesting advantage is that the hypothesis can be verified *with limited resources*. Unlike our alternative hypotheses, the original version can be verified without involving test subjects and controlled test environments, and without access to detailed – and sensitive – information on a given company’s operations. An advantage of e.g. our second alternative hypothesis is of course that it relates much more directly to the profit of company X.

In the following, we briefly cover the potentials and limitations of each of the four approaches to evaluation of DSL inventing projects.

The experimental approach This mode of evaluation requires that the evaluator has access to a realistic environment and the resources – usually in shape of time and people – for performing the experiments. This does not often seem to be the case.

When the requirements are fulfilled, an experimental evaluation may yield useful data not only about the DSL and its tools *an sich*, but also about the software development processes that they support, as was demonstrated in [KMB$^+$96]. Moreover, if the experiments meet scientific standards (as they did in [KMB$^+$96]), the confidence in their results should be very high.

A limitation to the approach is that we cannot really study organizations experimentally. Thus, if we are to perform an experimental evaluation, we must restrict our focus to DSL, tools and processes.

The formal approach This mode of evaluation may demand hard work and (sometimes) appropriate hardware and software for computer-supported verification, but it can be applied no matter what the organizational context is. There is no need for test subjects or sensitive information.

Development processes and organizations are arguably beyond the reach of strictly formal evaluations. But when a formal hypothesis about a DSL and its tools has been formulated, we may be able to establish its truth with extreme certainty: formal proof.

The organizational approach This mode of evaluation is applicable if the new DSL is used in an organization and the evaluator has some access to the records of this organization.[2]

The main strength of this approach is of course that it allows the evaluator to study the *actual* impact that a DSL had on an organization, not just its potential.

With the organizational approach, one must always assess the reliability of the available reports as evidence. For instance, a statement that DSL applications *seem* efficient may be doubtable if it is not backed by numbers. Even worse, members of the given organzation may have their own reasons for supporting or opposing the DSL, and may for that reason be questionable sources of information.

The engineering approach As with the formal approach, this mode of evaluation can be applied no matter what the organizational context is.

While a demonstration of the DSL and its tools does not give us facts regarding its direct impact on an organization using them, we may see it as a "prototype

[2]An exception is [Mog02], where the DSL was made available on the Internet, which led to a number of reactions from people that weren't related in an organizational sense.

Object of study:	1. DSL & tools	2. Dev. process	3. Organization
The experimental approach	$\surd$	$\surd$	
The formal approach	$\surd$		
The organizational approach	$\surd$	$\surd$	$\surd$
The engineering approach	$\surd$	$(\surd)$	

Figure 3.1: Applicability of the four approaches depending on the object of study.

	Requirements	Confidence
The experimental approach	Test subjects, test environment, people, time	High
The formal approach		Very high
The organizational approach	Access to organization records	Medium
The engineering approach		Medium

Figure 3.2: The left column indicates the contextual requirements to adopting each approach. The right column indicates the degree of confidence in your conclusions, each approach is fit to establish.

experiment" giving some evidence regarding the way the tools affect a development process. But clearly one must put less confidence in results obtained by the engineering approach than in results from controlled, scientific experiements.

3.5 Conclusion

We began by arguing that deciding to invent a DSL is an active choice, and therefore one that should be subjected to direct evaluation once the DSL inventing project has reached its final stages. We have reviewed a number of examples from the literature where DSL inventing projects were evaluated. The examples were classified into four categories based on how evaluation was carried out. We found that each of these four approaches had limited applicability, depending on the intended object of study, the context of the evaluation and how much confidence one is looking for. These findings are summed up in Figures 3.1 and 3.2.

While we have argued in favour of these classifications, we are aware that they leave much room for debate, and for more thorough literature studies. The present survey is intended as a basis for discussions, as we believe that the subject of evaluations needs to be treated in more depth in the DSL literature.

Part II

Partial evaluation theory

4 - Erlang semantics based on congruence

In this chapter we'll address the issue of a formal semantics for the concurrent programming language Erlang. Other definitions for such a semantics exists, but we found them insufficient for our ultimate purpose: to work with partial evaluation of Erlang programs (see Chapter 5). We therefore develop a somewhat differently styled semantics here.

The chapter develops the semantics in a number of steps to illustrate the decisions leading to the final design. In each step the syntax and/or semantics is extended somehow. On the way we also explore a couple of designs that *don't* work in order to illustrate the finer points of the Erlang definition. To avoid confusion we'll be quite verbose, stating the full set of rules each time. If the reader is only concerned with the final result, we refer to Figures 4.19 and 4.20.

Overall semantical style The semantics we develop is a structured operational semantics (SOS). Because of the nondeterminism and intended nontermination of parallel programs, this style of semantics is usually more convenient than e.g. natural semantics or denotational semantics. Indeed, it is not immediately clear what domain a denotational semantics should map to. The most obvious choice of domain is probably the set of expressions in the π-calculus ([Mil99]) or some other existing calculus which models parallelism ([Fou98, CG00, GHS02]). Of course, these are themselves associated with an operational semantics, so this would just be a reduction of the problem – we would be compiling Erlang to a language with a structured operational semantics. We will use methods from CCS ([Mil99]) and π-calculus to attack the problem of capturing certain aspects of Erlang program behaviour, in particular we shall use structural congruence to handle namespaces. Still, we found it more instructive to give an operational semantics directly rather than indirectly through compilation.

Overview of the chapter We begin by introducing the reader to Erlang. Following that, Section 4.2 restricts our attention to a small subset of the language and provides all the basic definitions behind our semantics. This results in a semantics where message transmission is not modelled in a precise way. Section 4.3 fixes this problem. The following sections add function calls, data structures and pattern matching to the syntax and semantics of the Erlang subset we consider. As mentioned, the final syntax and semantics are given in Figures 4.19 and 4.20. Section 4.7 comments on potential extensions of our semantics. Section 4.8 reviews existing alternative definitions of Erlang semantics. We conclude in Section 4.9

4.1 Introduction to Erlang

Erlang[1] is a language for writing stable, efficient and massively concurrent distributed systems, originally developed at Ericsson for programming telephone switches. Its history and main features are described in a very easily read paper by its main developer Joe Armstrong ([Arm97]). Erlang is available in an open-source format at *www.erlang.org*. The number of Erlang programmers has been estimated at 10,000 and it is used by several telecom utilities producers. It is a strict, higher-order functional language with a syntax inspired by *Prolog*.

Like most programming languages developed for a particular, practical purpose, Erlang has many constructs, features and libraries. A large number are described in the book [AVWW]. A substantial core part of Erlang, intended as a standard intermediate language in Erlang compilers, is described informally but meticulously in the *Core Erlang Specification* ([CGJ$^+$00]). Here we shall only introduce a small subset of Erlang, the syntax of which is given in Figure 4.1.

The reader with knowledge of Erlang may notice that we've focussed mainly on including the powerful primitives for spawning new processes, sending asynchronous messages between processes and receiving these messages using deep pattern matching.

Many of the language constructs are well-known from other functional languages and their semantics in Erlang is standard, but note that

- Curly brackets are used to construct (and destruct) tuples.
- The underscore pattern works as in *Prolog*: it matches *any* value.
- Comma denotes sequencing, so the meaning of $X = E_1, E_2$ is to evaluate E_1 to a value v, assign v to X and then evaluate E_2.

A value may be an atom (a lower-case term, e.g. **ok**), an integer, a tuple of values or a list of values. Each process is identified by a unique *process ID*. Such a process ID is also a value.

The really interesting productions are *spawn*, *self*, send (written E_1 ! E_2) and *receive*. We'll describe how these work in the following paragraphs.

[1]Named after Danish mathematician Agner Krarup Erlang, 1878 - 1929.

$$\begin{array}{rlcll}
\Gamma \in & \textit{Program} & ::= & \textit{Fundef}^{+} & \\
 & \textit{Fundef} & ::= & f(X_1,\ldots,X_n) \rightarrow \textit{Exp}. & ;\ n \geq 0 \\
E, E_1,\ldots,E_n \in & \textit{Exp} & ::= & X & \\
 & & \mid & n \ \mid \ E_1 + E_2 \ \mid \ E_1 - E_2 & \\
 & & \mid & a & \\
 & & \mid & X = E_1, E_2 & \\
 & & \mid & \texttt{case}\ E\ \texttt{of}\ M_1;\cdots;M_n\ \texttt{end} & ;\ n > 0 \\
 & & \mid & f(E_1,\ldots,E_n) & ;\ n \geq 0 \\
 & & \mid & [E_1,\ldots,E_n] & ;\ n \geq 0 \\
 & & \mid & [E_1 \mid E_2] & \\
 & & \mid & \{E_1,\ldots,E_n\} & ;\ n > 0 \\
 & & \mid & \texttt{spawn}(f,[E_1,\ldots,E_n]) & ;\ n \geq 0 \\
 & & \mid & \texttt{self}() & \\
 & & \mid & E_1\ !\ E_2 & \\
 & & \mid & \texttt{receive}\ M_1;\cdots;M_n\ \texttt{end} & ;\ n > 0 \\
M_1,\ldots,M_n \in & \textit{Match} & ::= & P \rightarrow E & \\
P \in & \textit{Pattern} & ::= & X & \\
 & & \mid & a & \\
 & & \mid & n & \\
 & & \mid & [P_1 \mid P_2] & \\
 & & \mid & \{P_1,\ldots,P_n\} & ;\ n > 0 \\
 & & \mid & \_ & \\
n \in & \mathbf{N} & & & \\
X, X_1,\ldots,X_n \in & \textit{Variables} & & & \\
f, a \in & \textit{Atoms} & & &
\end{array}$$

Figure 4.1: Syntax of an Erlang subset.

Spawn and self A spawn is like a function call, but instead of having the current process execute the call, a new process is constructed. This child process will run in parallel with the current, performing the function call. Spawn returns the process ID of the new process. If the function call terminates, the process executing it will end. If the function call returned a value, this value will be ignored.

We have simplified `spawn` a bit. In Erlang an extra parameter specifies the module in which the specified function is placed. We're not using modules in our subset, so we removed that parameter.

Messaging Processes can communicate through messages only. A message is simply a value, and any value can be a message. Unlike other concurrent frameworks, a message isn't sent through a named channel, but directly to a named process. Each process has its own *inbox* of messages, and any process can place a message (value) v in the inbox of process ρ by evaluating the expression $\rho\ !\ v$. Here ρ is the process ID of the receiving process. The send operator is asynchronous, so the sender process does not wait until the receiving process is ready to read the message. Neither does it wait for the message to be placed in the receiving process's inbox. The system does promise that all messages will eventually be delivered, and the messages from process ρ to process ρ' will be delivered in the same order they were sent ([AVWW], p. 69).

The inbox is a queue of messages. It may contain messages from many other processes (and from the inbox's owner process itself). The order of the messages reflects the order in which they have arrived. The inbox can only be accessed through the *receive* expression. The programmer specifies a number of (deep) patterns that can be matched to values in a natural way. Erlang tries to match the first message in the inbox to each of the patterns in sequence. If all fail to match, it proceeds to the next message and so on.

Receive is blocking – the process waits until a matching message has arrived, if there was not one in its inbox. In Erlang, this behaviour can be avoided using the *after* pattern, which specifies a maximum period for which the process may be blocked. We have not included this feature in our subset, though.

Missing Features Some important featues of Erlang are *not* included in our subset. These include:

- `when` guards (that add conditions on top of pattern matching),
- `after` pattern (which lets a program specify a maximal time to wait for a message),
- exceptions,
- `links` between processes (these links are used by the system to propagate a kill signal between sibling processes),
- higher-order expressions, and
- modules and hot code replacement.

$$
\begin{array}{rlcl}
E \in & \mathit{Exp} & ::= & X = \mathit{receive}\ \_,\ E \\
 & & \mid & X\ !\ Y,\ E \\
 & & \mid & X = \mathit{spawn}(f, [X_1, \ldots, X_n]),\ E \\
 & & \mid & X = \mathit{self}(),\ E \\
 & & \mid & \mathit{case} \diamond \mathit{of}\ \_ \to E;\ \_ \to E' \\
 & & \mid & X \\
n \in & \mathbf{N} & & \\
f \in & \mathit{Atoms} & & \\
X, X_1, \ldots, X_n \in & \mathit{Variables} & &
\end{array}
$$

Figure 4.2: The initial syntax we consider. The diamond denotes nondeterministic choice.

4.2 First version of the semantics

We start with a small syntax which is not quite an Erlang subset but rather an Erlang-like calculus, see Figure 4.2. The difference from real Erlang is the *case*-expression. Instead of worrying about pattern matching at this point, we simply make *case* a nondeterministic choice operator, denoted by the diamond and “don’t care” patterns in the syntax. We’ll add real pattern matching later.

To keep things simple, we don’t model the binding of function names (f in the syntax) to expressions. The semantics will expect an external system to handle that aspect. At the end of the chapter (Section 4.7) we’ll discuss how this could be incorporated in our final semantics.

The only values we can construct with the shown operators are process IDs, so for now we’ll assume that all values are process IDs:

$$\mathit{Values} = \mathit{ProcIDs}$$

The latter is an enumerable set of names analogous to the set of variable names is in λ-calculus. Elements of *Values* will be written

$$v, v', v_1, v_2, \ldots \in \mathit{Values}$$

When it is important that the value *is* a process ID, we’ll use the symbol ρ instead:

$$\rho, \rho', \rho_1, \rho_2, \ldots \in \mathit{ProcIDs}$$

The distinction will become important later on when we extend the set of values.

$$
\begin{array}{rlcl}
E \in & \mathit{Exp} & ::= & X = \mathit{receive}\ \_,\ E \\
 & & \mid & \delta_1\ !\ \delta_2,\ E \\
 & & \mid & X = \mathit{spawn}(f, [\delta_1, \ldots, \delta_n]),\ E \\
 & & \mid & X = \mathit{self}(),\ E \\
 & & \mid & \mathit{case} \diamond \mathit{of}\ \_ \to E;\ \_ \to E' \\
 & & \mid & \delta \\
n \in & \mathbf{N} & & \\
f \in & \mathit{Atoms} & & \\
\delta, \delta_1, \ldots, \delta_n \in & \mathit{Variables} \cup \mathit{Values} & &
\end{array}
$$

Figure 4.3: The syntax extended to allow a substitution semantics. Values can now be used in place of variable lookups.

We'll need a further feature not present in real Erlang: explicit process IDs in the code. In real Erlang, process IDs cannot be given as constants. But because of certain features in Erlang (e.g. deep patterns containing bound variables) we are aiming for a substitution semantics, rather than a semantics where a store of variable's values is used. For this reason we will need process IDs in the code during evaluation. The extended syntax is shown in Figure 4.3.

It is an error to assign the same variable X twice in a sequence of expressions. It is also an error to use the value of a variable that has not been assigned. Thus, the semantics will only have rules to evaluate

$$X = [\mathit{some\ right} - \mathit{hand\ side}], E$$

if there are no variables (only values) on the right-hand side of the assignment.

Before we give the first version of the semantics we shall need a final syntactic extension. It should not surprise the reader that we need to model sets of processes with associated process IDs. The *spawn* operator is the means for making new processes from existing, and it is exactly in the interaction between parallel processes that the interesting behaviour of Erlang programs occurs.

It is in the modelling of parallel runnning processes that we choose to play the game of CCS and π-calculus. Other definitions (see Section 4.8) of Erlang semantics have modelled this aspect using a finite set of processes (where a process is a triple composed of an expression, a process ID and a queue of messages). In e.g. CCS, the processes are placed in the leaves of a binary tree. An internal node in this tree is either

- the parallel composition operator, or

$$
\begin{array}{rlcl}
P \in & \mathit{Proc} & ::= & \langle \rho : E \rangle \\
 & & \mid & P \parallel P' \\
 & & \mid & \mathit{new}\ \rho\ P \\
 & & & \\
E \in & \mathit{Exp} & ::= & X = \mathit{receive}\ \_,\ E \\
 & & \mid & \delta_1\ !\ \delta_2,\ E \\
 & & \mid & X = \mathit{spawn}(f, [\delta_1, \ldots, \delta_n]),\ E \\
 & & \mid & X = \mathit{self}(),\ E \\
 & & \mid & \mathit{case} \diamond \mathit{of}\ \_ \to E;\ \_ \to E' \\
 & & \mid & \delta \\
n \in & \mathbf{N} & & \\
f \in & \mathit{Atoms} & & \\
\delta, \delta_1, \ldots, \delta_n \in & \mathit{Variables} \cup \mathit{Values} & &
\end{array}
$$

Figure 4.4: Syntax for parallel process composition.

- a *new*-operator which binds the name of a commmunication channel.

Erlang does not have communication channels in the sense of CCS, but we shall need the name-space handling for our process IDs. A significant difference between CCS and our semantics is that every expression has an associated process ID. Our version of the tree model is defined by the syntax in Figure 4.4. The process tree is itself viewed as process.

The real power of this tree model is that it comes with a notion of *structural congruence*. In general terms this means that the tree of processes may be reordered in any way, as long as name bindings are not captured or lost. In essence, the set of trees *modulo structural congruence* defines a space in which expressions reside. And in this space, a process is “next to” any other process when evaluation needs it to be, and possible name-space clashes are handled implicitly. The effect of this is an incredibly smooth integration between actions local to one process and global behaviour.

The structural congruence (henceforth just *congruence*) rules for our system are shown in Figure 4.5. The rules use two concepts that we have not formally defined: α-equivalence of processes (denoted by $=_\alpha$) and freeness of a process ID in a process. Please note first of all that these concepts do not refer to the variables (the Xs) in our expressions at all. The names we’re interested in handling on this level are always process IDs.

$$\frac{P \equiv P' \qquad P' \equiv P''}{P \equiv P''} \qquad\qquad \frac{P' \equiv P}{P \equiv P'}$$

$$\frac{P =_\alpha P'}{P \equiv P'}$$

$$\frac{}{P \| P' \equiv P' \| P}$$

$$\frac{}{P \| (P' \| P'') \equiv (P \| P') \| P''}$$

$$\frac{\rho \notin \mathit{fn}(P)}{\mathit{new}\ \rho\ (P \| P') \equiv P \| \mathit{new}\ \rho\ P'}$$

$$\frac{}{\mathit{new}\ \rho\rho'\ P \equiv \mathit{new}\ \rho'\rho\ P}$$

$$\frac{}{P \,\|\, \mathit{new}\ \rho\ \langle \rho : v \rangle \equiv P}$$

Figure 4.5: The definition of structural congruence.

The set of free process IDs of a process tree is given by

$$\begin{array}{lcl} \mathit{fn}(\langle \rho : E \rangle) & = & \{\rho\} \cup \mathit{procids}(E) \\ \mathit{fn}(P \| P') & = & \mathit{fn}(P) \cup \mathit{fn}(P') \\ \mathit{fn}(\mathit{new}\ \rho\ P) & = & \mathit{fn}(P) \setminus \{\rho\} \end{array}$$

The function *procids* simply returns the set of *all* process IDs that occur as sub-terms in the expression E it is applied to. An occurrence of a process ID ρ in a process is bound when it is not free, i.e. when an ancestor node in the process tree is of the form $\mathit{new}\ \rho\ P$. Two processes P and P' are alpha-equivalent, $P =_\alpha P'$, when P' can be obtained from P by renaming bound process IDs with fresh ones.

We shall sometimes use the term $(\mathit{new}\ \rho_1 \rho_2 \cdots \rho_n\ P)$ to mean

$$\mathit{new}\ \rho_1\ \mathit{new}\ \rho_2\ \cdots \mathit{new}\ \rho_n\ P$$

We also allow ambiguous sequences of parallel composition, e.g. $P \| P' \| P''$ as this operation is associative modulo congruence.

$$\frac{}{\langle \rho : X = \mathit{receive}\ \_, E\rangle \| \langle \rho' : \rho\ !\ v, E'\rangle \to \langle \rho : \{v/X\}E\rangle \| \langle \rho' : E'\rangle}\ \mathsf{React-Sync}$$

$$\frac{f(X_1,\ldots,X_n) \triangleq E_f \quad \rho' \notin fn(\langle \rho : E\rangle) \quad P = \langle \rho' : \{v_1/X_1\}\cdots\{v_n/X_n\}E_f\rangle}{\langle \rho : X = \mathit{spawn}(f,[v_1,\ldots,v_n]),E\rangle \to \mathit{new}\ \rho'\ (\langle \rho : \{\rho'/X\}E\rangle \| P)}\ \mathsf{Spawn}$$

$$\frac{}{\langle \rho : X = \mathit{self}(),E\rangle \to \langle \rho : \{\rho/X\}E\rangle}\ \mathsf{Self}$$

$$\frac{i \in \{1,2\}}{\langle \rho : \mathit{case} \diamond \mathit{of}\ \_ \to E_1;\ \_ \to E_2\rangle \to \langle \rho : E_i\rangle}\ \mathsf{Case}$$

$$\frac{P \to P''}{P\|P' \to P''\|P'}\ \mathsf{Par}$$

$$\frac{P \to P'}{\mathit{new}\ \rho\ P \to \mathit{new}\ \rho\ P'}\ \mathsf{Res}$$

$$\frac{P_1 \to P_2 \quad P_1 \equiv P_1' \quad P_2 \equiv P_2'}{P_1' \to P_2'}\ \mathsf{Struct}$$

Figure 4.6: The first version of the semantics. In this version the interaction is synchronous.

The last congruence rule may look odd. Its effect is simply to allow the removal of any process that has ended, which in our case must mean it matches the last production of the expression syntax (and that the variable in that production has been substituted by a value). Note that we cannot remove a process if another process still has a reference to it (i.e. its process ID), because the latter process would have to appear under the binding of the former process's ID, preventing the application of the last congruence rule. Bad programming style may thus cause a lot of finished processes to remain in the tree.

We are now ready to give the first version of our operational semantics. It is shown in Figure 4.6. Note that the result of substituting a variable X with its value v in an expression E is denoted $\{v/X\}E$.

The semantics is given in the form of a number of *reaction rules*. The first one concerns the transmission of messages. It states that if two processes are next to each other in the process tree, if the first one is waiting for a message, and if the second one has a message for it, the message is transmitted. This is synchronous communication, but we'll fix that in a short while.

Note how the three last rules make sure that the actual arrangement of processes in the tree is not important – if one process is waiting for a message and another has one to send, they *can* react. This was the effect we described above.

The Spawn rule shows how a new process is generated. The $\stackrel{\triangle}{=}$ operator queries the environment for the definition of function f. This is where our semantics implicitly depends on a mapping of function names to expressions. Note that the process ID of the new process can be chosen locally in the subtree containing only its parent process. The congruence relation will sort out any name clashes. We assume (as is reasonable) that function definitions do not contain explicit values, in particular process IDs.

The *self()* and *case* rules should contain no surprises.

4.3 Getting synchronicity right

As mentioned, the semantics is synchronous, which is not what Erlang implements. One way of making it asynchronous is to follow in the footsteps of asynchronous π-calculus. In that system, messages can flow around in the process tree along with the processes. This allows the sending process to "let go" of its message at any time, basically spawning it as a sibling process that will go looking for its receiver.

So let us for a moment assume that we add the following production for non-terminal P:

$$P ::= [\overline{\rho} v]$$

We could then split the React-rule into two rules to achieve asynchronous communication, see Figure 4.7.

There is a problem with this approach, however. Messages from one given process to another may switch order while flowing from sender to receiver, so for example:

$$\begin{gathered} new\ \rho_1\rho_2\rho_3\rho_4\ (\langle\rho_1 : \rho_2\ !\ \rho_3, \rho_2\ !\ \rho_4, E_1\rangle \| \langle\rho_2 : X = receive\ \_, E_2\rangle) \\ \rightarrow^* \\ new\ \rho_1\rho_2\rho_3\rho_4\ (\langle\rho_1 : E_1\rangle \| [\overline{\rho_2}\rho_3] \| \langle\rho_2 : \{\rho_4/X\}E_2\rangle) \end{gathered}$$

which is not how Erlang works. Message order is kept between two given processes. We are therefore forced to introduce a different mechanism to model the flow of messages. This mechanism is the Erlang inbox, see Figure 4.8. There is one inbox for each process ρ, named $inbox_\rho()$.

$$\frac{}{\langle \rho' : \rho \;!\; v, E\rangle \to \langle \rho' : E\rangle \| [\overline{\rho} v]} \quad \textsf{React} - \textsf{ASync1}$$

$$\frac{}{\langle \rho : X = receive\ \_, E\rangle \| [\overline{\rho} v] \to \langle \rho : \{v/X\}E\rangle} \quad \textsf{React} - \textsf{Async2}$$

$$\frac{f(X_1, \ldots, X_n) \stackrel{\triangle}{=} E_f \qquad \rho' \notin fn(\langle \rho : E\rangle) \qquad P = \langle \rho' : \{v_1/X_1\} \cdots \{v_n/X_n\}E_f\rangle}{\langle \rho : X = spawn(f, [v_1, \ldots, v_n]), E\rangle \to new\ \rho'\ (\langle \rho : \{\rho'/X\}E\rangle \| P)} \quad \textsf{Spawn}$$

$$\frac{}{\langle \rho : X = self(), E\rangle \to \langle \rho : \{\rho/X\}E\rangle} \quad \textsf{Self}$$

$$\frac{i \in \{1, 2\}}{\langle \rho : case \diamond of\ \_ \to E_1;\ \_ \to E_2\rangle \to \langle \rho : E_i\rangle} \quad \textsf{Case}$$

$$\frac{P \to P''}{P \| P' \to P'' \| P'} \quad \textsf{Par}$$

$$\frac{P \to P'}{new\ \rho\ P \to new\ \rho\ P'} \quad \textsf{Res}$$

$$\frac{P_1 \to P_2 \qquad P_1 \equiv P_1' \qquad P_2 \equiv P_2'}{P_1' \to P_2'} \quad \textsf{Struct}$$

Figure 4.7: Semantics in the style of asynchronus π-calculus. The syntax for a process here includes a construct for messages in transfer.

$$\begin{array}{llcl}
P \in & Proc & ::= & \langle \rho : E\rangle \\
 & & \mid & P \parallel P' \\
 & & \mid & new\ \rho\ P \\
 & & \mid & inbox_{\rho}(v_1 \cdots v_n) \\
 & & & \\
n \in & \mathbf{N} & &
\end{array}$$

Figure 4.8: The syntax extended with Erlang's inboxes.

An inbox can keep any number of messages to one given process and it keeps track of the order in which they arrived. This mechanism is applied in the semantics in Figure 4.9.

$$\frac{}{\langle \rho' : \rho \,!\, v, E\rangle \| \mathit{inbox}_{\rho}(v_1 \cdots v_n) \rightarrow \langle \rho' : E\rangle \| \mathit{inbox}_{\rho}(v_1 \cdots v_n v)} \quad \textsf{React}-\textsf{Inbox1}$$

$$\frac{}{\langle \rho : X = \mathit{receive}\ \_, E\rangle \| \mathit{inbox}_{\rho}(v_1 \cdots v_n) \rightarrow \langle \rho : \{v_1/X\}E\rangle \| \mathit{inbox}_{\rho}(v_2 \cdots v_n)} \quad \textsf{React}-\textsf{Inbox2}$$

$$\frac{f(X_1,\ldots,X_n) \overset{\triangle}{=} E_f \qquad \rho' \notin \mathit{fn}(\langle \rho : E\rangle) \qquad P = \langle \rho' : \{v_1/X_1\}\cdots\{v_n/X_n\}E_f\rangle}{\langle \rho : X = \mathit{spawn}(f,[v_1,\ldots,v_n]), E\rangle \rightarrow \mathit{new}\ \rho'\ (\langle \rho : \{\rho'/X\}E\rangle \| P \| \mathit{inbox}_{\rho'}())} \quad \textsf{Spawn}$$

$$\frac{}{\langle \rho : X = \mathit{self}(), E\rangle \rightarrow \langle \rho : \{\rho/X\}E\rangle} \quad \textsf{Self}$$

$$\frac{i \in \{1,2\}}{\langle \rho : \mathit{case} \diamond \mathit{of}\ \_ \rightarrow E_1;\ \_ \rightarrow E_2\rangle \rightarrow \langle \rho : E_i\rangle} \quad \textsf{Case}$$

$$\frac{P \rightarrow P''}{P \| P' \rightarrow P'' \| P'} \quad \textsf{Par}$$

$$\frac{P \rightarrow P'}{\mathit{new}\ \rho\ P \rightarrow \mathit{new}\ \rho\ P'} \quad \textsf{Res}$$

$$\frac{P_1 \rightarrow P_2 \qquad P_1 \equiv P_1' \qquad P_2 \equiv P_2'}{P_1' \rightarrow P_2'} \quad \textsf{Struct}$$

Figure 4.9: An asynchronous semantics where messages cannot switch order.

A sender process now transmits its message through the receiver's inbox. Messages cannot switch order on the way. Of course, we need to spawn an inbox with each new process as illustrated in the Spawn rule. We should also eliminate the inbox of any process removed by the congruence relation. The last rule of congruence should now read as follows:

$$\frac{}{P \| \mathit{new}\ \rho\ (\langle \rho : v\rangle \| \mathit{inbox}_{\rho}(v_1 \cdots v_n)) \equiv P}$$

$$\begin{array}{rlll} P \in & Proc & ::= & \langle \rho : E \rangle \\ & & | & P \parallel P' \\ & & | & new\ \rho\ P \\ & & | & inbox_\rho(v_1 \cdots v_n) \\ & & | & ether_\rho(v_1 \cdots v_n) \\ \\ n \in & \mathbf{N} & & \end{array}$$

Figure 4.10: The syntax extended with a process-specific "ether channel".

There is still a problem, although it is much more subtle than the last one. The problem is that unlike real Erlang, we cannot reach a state where A has sent a message to B (and moved on) but the message is not in B's inbox to inspect; it is still flowing in the "ether" towards B's inbox. This will be an issue when we introduce a more powerful *receive* operator.

We choose a solution that fits the same picture as the introduction of inboxes: we add an "ether channel" for each process, see Figure 4.10.

The ether will receive messages for its owner, then pass them along (in order!) to the inbox. While the messages are in the ether, they cannot be inspected. The semantics is given in Figure 4.11.

We have extended the Spawn rule to spawn both an ether and an inbox with each new process, and we also need to update the last rule of the congruence definition to

$$\frac{}{P \| new\ \rho\ (\langle \rho : v \rangle \| inbox_\rho(v_1 \cdots v_n) \| ether_\rho(v'_1 \cdots v'_m)) \equiv P}$$

The reader may have realized that we could get rid of the inbox-construct now we have the ether. This could be achieved by instead moving the queue of messages in the inbox *inside* the $\langle \rho : E \rangle$ pair (making it a triple). We choose to keep the current form because it is more modular; rules that don't concern the inbox (e.g. Case) can and do ignore it.

4.4 Function calls

With messages working as intended, we're ready to move towards a more complete functional language with more powerful operators (and without nondeter-

$$\frac{}{\langle \rho' : \rho \,!\, v, E\rangle \| \mathit{ether}_\rho(v_1 \cdots v_n) \to \langle \rho' : E\rangle \| \mathit{ether}_\rho(v_1 \cdots v_n v)}\ \textsf{React1}$$

$$\frac{}{\mathit{ether}_\rho(v_1 \cdots v_n) \| \mathit{inbox}_\rho(v'_1 \cdots v'_m) \to \mathit{ether}_\rho(v_2 \cdots v_n) \| \mathit{inbox}_\rho(v'_1 \cdots v'_m v_1)}\ \textsf{React2}$$

$$\frac{}{\langle \rho : X = \mathit{receive}\ \_, E\rangle \| \mathit{inbox}_\rho(v_1 \cdots v_n) \to \langle \rho : \{v_1/X\}E\rangle \| \mathit{inbox}_\rho(v_2 \cdots v_n)}\ \textsf{React3} - \textsf{Any}$$

$$\frac{f(X_1,\ldots,X_n) \stackrel{\triangle}{=} E_f \qquad \rho' \notin \mathit{fn}(\langle \rho : E\rangle) \qquad P = \langle \rho' : \{v_1/X_1\} \cdots \{v_n/X_n\}E_f\rangle}{\langle \rho : X = \mathit{spawn}(f, [v_1,\ldots,v_n]), E\rangle \to \mathit{new}\ \rho'\ (\langle \rho : \{\rho'/X\}E\rangle \| P \| \mathit{inbox}_{\rho'}() \| \mathit{ether}_{\rho'}())}\ \textsf{Spawn}$$

$$\frac{}{\langle \rho : X = \mathit{self}(), E\rangle \to \langle \rho : \{\rho/X\}E\rangle}\ \textsf{Self}$$

$$\frac{i \in \{1,2\}}{\langle \rho : \mathit{case} \diamond \mathit{of}\ \_ \to E_1;\ \_ \to E_2\rangle \to \langle \rho : E_i\rangle}\ \textsf{Case}$$

$$\frac{P \to P''}{P \| P' \to P'' \| P'}\ \textsf{Par}$$

$$\frac{P \to P'}{\mathit{new}\ \rho\ P \to \mathit{new}\ \rho\ P'}\ \textsf{Res}$$

$$\frac{P_1 \to P_2 \qquad P_1 \equiv P'_1 \qquad P_2 \equiv P'_2}{P'_1 \to P'_2}\ \textsf{Struct}$$

Figure 4.11: An asynchronous semantics where messages can be “in the ether”.

ministic choice). The first step is function calls with the syntax given in Figure 4.12.

There are several ways to go about adding calls to the semantics. But we should beware that one of the standard approaches is *not* an option in this case. We might think to implement function calls using the transitive extension of the evaluation relation, as in

$$\frac{f(X_1,\ldots,X_n) \stackrel{\triangle}{=} E_f \qquad \langle \rho : \{v_1/X_1\} \cdots \{v_n/X_n\}E_f\rangle \to^* \langle \rho : v'\rangle}{\langle \rho : X = f(v_1,\ldots,v_n), E\rangle \to \langle \rho : \{v'/X\}E\rangle}$$

$$
\begin{array}{rlcl}
E \in & \mathit{Exp} & ::= & X = \mathit{receive}\ \_,\ E \\
 & & \mid & \delta_1\ !\ \delta_2,\ E \\
 & & \mid & X = \mathit{spawn}(f, [\delta_1, \ldots, \delta_n]),\ E \\
 & & \mid & X = \mathit{self}(),\ E \\
 & & \mid & \mathit{case} \diamond \mathit{of}\ \_ \rightarrow E;\ \_ \rightarrow E' \\
 & & \mid & \delta \\
 & & \mid & X = f(\delta_1, \ldots, \delta_n), E \\
n \in & \mathbf{N} & & \\
f \in & \mathit{Atoms} & & \\
\delta, \delta_1, \ldots, \delta_n \in & \mathit{Variables} \cup \mathit{Values} & &
\end{array}
$$

Figure 4.12: Expression syntax with function calls.

$$
\begin{array}{rlcl}
P \in & \mathit{Proc} & ::= & \langle \rho : E \rangle \\
 & & \mid & P \parallel P' \\
 & & \mid & \mathit{new}\ \rho\ P \\
 & & \mid & \mathit{inbox}_\rho (v_1 \cdots v_n) \\
 & & \mid & \mathit{ether}_\rho (v_1 \cdots v_n) \\
 & & \mid & \mathit{stack}_\rho ((X_1, E_1) \cdots (X_n, E_n)) \\
 & & & \\
n \in & \mathbf{N} & &
\end{array}
$$

Figure 4.13: Process syntax with call stacks external to processes.

This does not work in a parallel language like Erlang, because the premise expects the function body to be evaluable *on its own*, without other processes that might interact with it.

A completely valid option – and the one taken by other semantics definitions for Erlang (see Section 4.8) – is to replace the expression part of the $\langle \rho : E \rangle$ pair by a call stack, thus keeping the stack explicit in the process.

We choose to go in another direction. Appreciating the modularity we obtained by keeping the inbox external to the $\langle \rho : E \rangle$ pair, we use the same method for hiding the call stack from rules that it does not concern. The resulting syntax is shown in Figure 4.13. We use variable L to range over lists of variable/expression pairs.

$$\frac{}{\langle \rho' : \rho\ !\ v, E\rangle \| ether_{\rho}(v_1 \cdots v_n) \to \langle \rho' : E\rangle \| ether_{\rho}(v_1 \cdots v_n v)}\ \textsf{React1}$$

$$\frac{}{ether_{\rho}(v_1 \cdots v_n) \| inbox_{\rho}(v'_1 \cdots v'_m) \to ether_{\rho}(v_2 \cdots v_n) \| inbox_{\rho}(v'_1 \cdots v'_m v_1)}\ \textsf{React2}$$

$$\frac{}{\langle \rho : X = receive\ \_, E\rangle \| inbox_{\rho}(v_1 \cdots v_n) \to \langle \rho : \{v_1/X\}E\rangle \| inbox_{\rho}(v_2 \cdots v_n)}\ \textsf{React3 − Any}$$

$$\frac{f(X_1,\ldots,X_n) \triangleq E_f \quad \rho' \notin fn(\langle \rho : E\rangle) \quad P = \langle \rho' : \{v_1/X_1\}\cdots\{v_n/X_n\}E_f\rangle}{\langle \rho : X = spawn(f,[v_1,\ldots,v_n]), E\rangle \to new\ \rho'\ (\langle \rho : \{\rho'/X\}E\rangle \| P \| inbox_{\rho'}() \| ether_{\rho'}() \| stack_{\rho'}())}\ \textsf{Spawn}$$

$$\frac{}{\langle \rho : X = self(), E\rangle \to \langle \rho : \{\rho/X\}E\rangle}\ \textsf{Self}$$

$$\frac{i \in \{1,2\}}{\langle \rho : case \diamond of\ \_ \to E_1;\ \_ \to E_2\rangle \to \langle \rho : E_i\rangle}\ \textsf{Case}$$

$$\frac{f(X_1,\ldots,X_n) \triangleq E_f \quad E' = \{v_1/X_1\}\cdots\{v_n/X_n\}E_f}{\langle \rho : X = f(v_1,\ldots,v_n), E\rangle \| stack_{\rho}(L) \to \langle \rho : E'\rangle \| stack_{\rho}((X,E)\ L)}\ \textsf{Call}$$

$$\frac{}{\langle \rho : v\rangle \| stack_{\rho}((X,E)\ L) \to \langle \rho : \{v/X\}E\rangle \| stack_{\rho}(L)}\ \textsf{Return}$$

$$\frac{P \to P''}{P \| P' \to P'' \| P'}\ \textsf{Par}$$

$$\frac{P \to P'}{new\ \rho\ P \to new\ \rho\ P'}\ \textsf{Res}$$

$$\frac{P_1 \to P_2 \quad P_1 \equiv P'_1 \quad P_2 \equiv P'_2}{P'_1 \to P'_2}\ \textsf{Struct}$$

Figure 4.14: The semantics with function call and return.

Given this choice, the call semantics is easy to define (see Figure 4.14). Note that expressions that are evaluated down to a value are now important and may not be removed by process congruence, because they represent return values. The exception is when the call stack is empty:

$$\frac{}{P \parallel new\ \rho\ (\langle \rho : v\rangle \| inbox_{\rho}(v_1 \cdots v_n) \| ether_{\rho}(v'_1 \cdots v'_m) \| stack_{\rho}()) \ \equiv\ P}$$

$$
\begin{array}{rlcl}
E \in & \mathit{Exp} & ::= & X = \mathit{receive}\ \_,\ E \\
 & & \mid & \delta_1\ !\ \delta_2,\ E \\
 & & \mid & X = \mathit{spawn}(f, [\delta_1, \ldots, \delta_n]),\ E \\
 & & \mid & X = \mathit{self}(),\ E \\
 & & \mid & \mathit{case} \diamond \mathit{of}\ \_ \to E;\ \_ \to E' \\
 & & \mid & \delta \\
 & & \mid & X = f(\delta_1, \ldots, \delta_n), E \\
 & & \mid & X = [], E \\
 & & \mid & X = [\delta_1 | \delta_2], E \\
n \in & \mathbf{N} & & \\
f \in & \mathit{Atoms} & & \\
\delta, \delta_1, \ldots, \delta_n \in & \mathit{Variables} \cup \mathit{Values} & &
\end{array}
$$

Figure 4.15: Expression syntax with list constructors added.

4.5 Adding lists

The next step is enriching the value domain so that we can get closer to real programming. We'll extend the current value domain with lists. Extending with tuples and integers would be done in an equivalent manner, but we leave those to the reader.

In Erlang, the list constructors are syntactically written as $[]$ and $[E \mid E']$. We shall need a notation for referring to list *values* that preferably differs from Erlang syntax. We choose to let our list *value* constructors have Lisp-like syntax, so that the value associated with the expression $[]$ is denoted by *nil* and the value associated with the expression $[E \mid E']$ is denoted $\mathit{cons}\ v\ v'$, when v and v' are the values of E and E' respectively. Thus we define the set of values recursively by

$$V \in \mathit{Values} = \mathit{ProcIDs} \cup \{\mathit{nil}\} \cup \{\mathit{cons}\ v_1\ v_2 \mid v_1, v_2 \in \mathit{Values}\}$$

We say that a value v is *atomic* and write $\mathit{atom}(v)$ if and only if $v \in \mathit{ProcIDs} \cup \{\mathit{nil}\}$.

To be able to construct lists we add two productions for expressions, see Figure 4.15. The semantics of the constructors is simple as can be, and we have added the rules to the system in Figure 4.16.

4.6 Patterns and case

Having a more interesting set of values, we're ready to define a more expressive version of *receive*. The Erlang *receive* can do three jobs in one operation:

$$\frac{}{\langle \rho' : \rho \,!\, v, E\rangle \| \mathit{ether}_{\rho}(v_1 \cdots v_n) \rightarrow \langle \rho' : E\rangle \| \mathit{ether}_{\rho}(v_1 \cdots v_n v)} \quad \textsf{React1}$$

$$\frac{}{\mathit{ether}_{\rho}(v_1 \cdots v_n) \| \mathit{inbox}_{\rho}(v'_1 \cdots v'_m) \rightarrow \mathit{ether}_{\rho}(v_2 \cdots v_n) \| \mathit{inbox}_{\rho}(v'_1 \cdots v'_m v_1)} \quad \textsf{React2}$$

$$\frac{}{\langle \rho : X = \mathit{receive}\ \_, E\rangle \| \mathit{inbox}_{\rho}(v_1 \cdots v_n) \rightarrow \langle \rho : \{v_1/X\}E\rangle \| \mathit{inbox}_{\rho}(v_2 \cdots v_n)} \quad \textsf{React3} - \textsf{Any}$$

$$\frac{f(X_1,\ldots,X_n) \triangleq E_f \qquad \rho' \notin \mathit{fn}(\langle \rho : E\rangle) \qquad P = \langle \rho' : \{v_1/X_1\} \cdots \{v_n/X_n\}E_f\rangle}{\langle \rho : X = \mathit{spawn}(f,[v_1,\ldots,v_n]),E\rangle \rightarrow \mathit{new}\ \rho'\ (\langle \rho : \{\rho'/X\}E\rangle \| P \| \mathit{inbox}_{\rho'}() \| \mathit{ether}_{\rho'}() \| \mathit{stack}_{\rho'}())} \quad \textsf{Spawn}$$

$$\frac{}{\langle \rho : X = \mathit{self}(),E\rangle \rightarrow \langle \rho : \{\rho/X\}E\rangle} \quad \textsf{Self}$$

$$\frac{i \in \{1,2\}}{\langle \rho : \mathit{case} \diamond \mathit{of}\ \_ \rightarrow E_1;\ \_ \rightarrow E_2\rangle \rightarrow \langle \rho : E_i\rangle} \quad \textsf{Case}$$

$$\frac{f(X_1,\ldots,X_n) \triangleq E_f \qquad E' = \{v_1/X_1\} \cdots \{v_n/X_n\}E_f}{\langle \rho : X = f(v_1,\ldots,v_n),E\rangle \| \mathit{stack}_{\rho}(L) \rightarrow \langle \rho : E'\rangle \| \mathit{stack}_{\rho}((X,E)\ L)} \quad \textsf{Call}$$

$$\frac{}{\langle \rho : v\rangle \| \mathit{stack}_{\rho}((X,E)\ L) \rightarrow \langle \rho : \{v/X\}E\rangle \| \mathit{stack}_{\rho}(L)} \quad \textsf{Return}$$

$$\frac{}{\langle \rho : X = [],E\rangle \rightarrow \langle \rho : \{\mathit{nil}/X\}E\rangle} \quad \textsf{Nil}$$

$$\frac{}{\langle \rho : X = [v_1|v_2],E\rangle \rightarrow \langle \rho : \{\mathit{cons}\ v_1\ v_2/X\}E\rangle} \quad \textsf{Cons}$$

$$\frac{P \rightarrow P''}{P \| P' \rightarrow P'' \| P'} \quad \textsf{Par}$$

$$\frac{P \rightarrow P'}{\mathit{new}\ \rho\ P \rightarrow \mathit{new}\ \rho\ P'} \quad \textsf{Res}$$

$$\frac{P_1 \rightarrow P_2 \qquad P_1 \equiv P'_1 \qquad P_2 \equiv P'_2}{P'_1 \rightarrow P'_2} \quad \textsf{Struct}$$

Figure 4.16: List constructor semantics added. Note that *nil* and *cons* operate in the value domain – not in Erlang itself.

$$
\begin{array}{rlcl}
E \in & \mathit{Exp} & ::= & X = \mathit{receive}\ M,\ E \\
& & \mid & \delta_1\ !\ \delta_2,\ E \\
& & \mid & X = \mathit{spawn}(f, [\delta_1, \ldots, \delta_n]),\ E \\
& & \mid & X = \mathit{self}(),\ E \\
& & \mid & \mathit{case} \diamond \mathit{of}\ \_ \rightarrow E;\ \_ \rightarrow E' \\
& & \mid & \delta \\
& & \mid & X = f(\delta_1, \ldots, \delta_n), E \\
& & \mid & X = [], E \\
& & \mid & X = [\delta_1 | \delta_2], E \\
\\
M \in & \mathit{Match} & ::= & \_ \\
& & \mid & \delta \\
& & \mid & [] \\
& & \mid & [M_1 | M_2] \\
& & \mid & M_1; \cdots; M_n \\
n \in & \mathbf{N} & & \\
f \in & \mathit{Atoms} & & \\
\delta, \delta_1, \ldots, \delta_n \in & \mathit{Variables} \cup \mathit{Values} & &
\end{array}
$$

Figure 4.17: Syntax with deep patterns in the receive-operator.

- Specify what kind of message should be taken from the inbox,
- destruct lists and other structures, and
- branch control depending on the structure of the picked message.

We'll extend *case* to be able to handle the two latter tasks and focus solely on the first feature. We shall need a syntax for patterns to match the inbox messages against, see Figure 4.17.

For each pattern we need to know which messages (values) match it. This is specified by the function $\mathit{matches}(\cdot) : \mathit{Match} \rightarrow \mathcal{P}(\mathit{Values})$, given by

$$
\begin{array}{lcl}
\mathit{matches}(\_) & = & \mathit{Values} \\
\mathit{matches}(v) & = & \{v\} \\
\mathit{matches}([]) & = & \{\mathit{nil}\} \\
\mathit{matches}([M_1|M_2]) & = & \{\mathit{cons}\ v_1\ v_2 \mid v_1 \in \mathit{matches}(M_1) \wedge v_2 \in \mathit{matches}(M_2)\} \\
\mathit{matches}(M_1; \cdots; M_n) & = & \mathit{matches}(M_1) \cup \cdots \cup \mathit{matches}(M_n)
\end{array}
$$

There is no rule for $\mathit{matches}(X)$. Such a pattern would represent a destructor and we will not allow that in the *receive*. This represents a slight simplification, as we cannot pick e.g. the first message that is a 2-tuple with identical elements. We can now put the third reaction rule on its final form, see Figure 4.18.

The final step in defining our Erlang semantics regards *case*, implementing both branching and destruction (of lists). We give separate productions for these two uses of *case* in order to make the semantics simpler. The final syntax of our Erlang subset is given in Figure 4.19.

The semantics for the branching *case* is uncomplicated. The deconstruction of lists needs to assign two variables, so note that none of those may be previously defined. The final semantics is given in Figure 4.20.

4.7 Final notes on our semantics

We'll end by making a few observations that may be useful if you would like to use or extend the system.

Initial state We have not discussed what the initial state of a program's execution is. There are a number of reasonable choices for this; we simply choose to define

$$\mathit{InitialState}(\rho, E) = \langle \rho : E \rangle \| \mathit{stack}_\rho() \| \mathit{inbox}_\rho() \| \mathit{ether}_\rho()$$

So the initial state is defined relative to a process ID ρ and an initial expression E. The program itself, i.e. the set Γ of function definitions, is implicit. When the relevant Γ is not clear from the context, we will make it explicit as a subscript to the semantical relation $\to$, so that

$$P \to_\Gamma P'$$

means "P reacts to P' when Γ is the set of function definitions".

Ending match lists with "end" In real Erlang, both *receive* and *case* must end with the keyword *end*. We have saved a bit of space by skipping this syntactical detail.

Evaluation order The subset of Erlang we have described only allow expressions that are linear in their structure, in the sense that a composite expression like

$$X = f(E_1, E_2), E$$

$$\frac{}{\langle \rho' : \rho\ !\ v, E\rangle \| ether_{\rho}(v_1 \cdots v_n) \rightarrow \langle \rho' : E\rangle \| ether_{\rho}(v_1 \cdots v_n v)} \quad \textsf{React1}$$

$$\frac{}{ether_{\rho}(v_1 \cdots v_n) \| inbox_{\rho}(v'_1 \cdots v'_m) \rightarrow ether_{\rho}(v_2 \cdots v_n) \| inbox_{\rho}(v'_1 \cdots v'_m v_1)} \quad \textsf{React2}$$

$$\frac{k = \min\{i \mid v_i \in matches(M)\}}{\langle \rho : X = receive\ M, E\rangle \| inbox_{\rho}(v_1 \cdots v_n) \rightarrow \langle \rho : \{v_k/X\}E\rangle \| inbox_{\rho}(v_1 \cdots v_{k-1} v_{k+1} \cdots v_n)} \quad \textsf{React3}$$

$$\frac{f(X_1, \ldots, X_n) \stackrel{\triangle}{=} E_f \qquad \rho' \notin fn(\langle \rho : E\rangle) \qquad P = \langle \rho' : \{v_1/X_1\} \cdots \{v_n/X_n\} E_f\rangle}{\langle \rho : X = spawn(f, [v_1, \ldots, v_n]), E\rangle \rightarrow new\ \rho'\ (\langle \rho : \{\rho'/X\}E\rangle \| P \| inbox_{\rho'}() \| ether_{\rho'}() \| stack_{\rho'}())} \quad \textsf{Spawn}$$

$$\frac{}{\langle \rho : X = self(), E\rangle \rightarrow \langle \rho : \{\rho/X\}E\rangle} \quad \textsf{Self}$$

$$\frac{i \in \{1, 2\}}{\langle \rho : case \diamond of\ \_ \rightarrow E_1;\ \_ \rightarrow E_2\rangle \rightarrow \langle \rho : E_i\rangle} \quad \textsf{Case}$$

$$\frac{f(X_1, \ldots, X_n) \stackrel{\triangle}{=} E_f \qquad E' = \{v_1/X_1\} \cdots \{v_n/X_n\} E_f}{\langle \rho : X = f(v_1, \ldots, v_n), E\rangle \| stack_{\rho}(L) \rightarrow \langle \rho : E'\rangle \| stack_{\rho}((X, E)\ L)} \quad \textsf{Call}$$

$$\frac{}{\langle \rho : v\rangle \| stack_{\rho}((X, E)\ L) \rightarrow \langle \rho : \{v/X\}E\rangle \| stack_{\rho}(L)} \quad \textsf{Return}$$

$$\frac{}{\langle \rho : X = [], E\rangle \rightarrow \langle \rho : \{nil/X\}E\rangle} \quad \textsf{Nil}$$

$$\frac{}{\langle \rho : X = [v_1 | v_2], E\rangle \rightarrow \langle \rho : \{cons\ v_1\ v_2/X\}E\rangle} \quad \textsf{Cons}$$

$$\frac{P \rightarrow P''}{P \| P' \rightarrow P'' \| P'} \quad \textsf{Par}$$

$$\frac{P \rightarrow P'}{new\ \rho\ P \rightarrow new\ \rho\ P'} \quad \textsf{Res}$$

$$\frac{P_1 \rightarrow P_2 \qquad P_1 \equiv P'_1 \qquad P_2 \equiv P'_2}{P'_1 \rightarrow P'_2} \quad \textsf{Struct}$$

Figure 4.18: Semantics implementing pattern matching.

$$
\begin{array}{rlcl}
P \in & \mathit{Proc} & ::= & \langle \rho : E \rangle \\
 & & \mid & P \parallel P' \\
 & & \mid & \mathit{new}\ \rho\ P \\
 & & \mid & \mathit{inbox}_\rho(v_1 \cdots v_n) \\
 & & \mid & \mathit{ether}_\rho(v_1 \cdots v_n) \\
 & & \mid & \mathit{stack}_\rho((X_1, E_1) \cdots (X_n, E_n)) \\
 & & & \\
E \in & \mathit{Exp} & ::= & X = \mathit{receive}\ M,\ E \\
 & & \mid & \delta_1\ !\ \delta_2,\ E \\
 & & \mid & X = \mathit{spawn}(f, [\delta_1, \ldots, \delta_n]),\ E \\
 & & \mid & X = \mathit{self}(),\ E \\
 & & \mid & \mathit{case}\ \delta\ \mathit{of}\ \delta' \to E;\ \_ \to E' \\
 & & \mid & \mathit{case}\ \delta\ \mathit{of}\ [X|X'] \to E;\ \_ \to E' \\
 & & \mid & \delta \\
 & & \mid & X = f(\delta_1, \ldots, \delta_n), E \\
 & & \mid & X = [], E \\
 & & \mid & X = [\delta_1|\delta_2], E \\
 & & & \\
M \in & \mathit{Match} & ::= & \_ \\
 & & \mid & \delta \\
 & & \mid & [] \\
 & & \mid & [M_1|M_2] \\
 & & \mid & M_1; \cdots; M_n \\
n \in & \mathbf{N} & & \\
f \in & \mathit{Atoms} & & \\
\delta, \delta_1, \ldots, \delta_n \in & \mathit{Variables} \cup \mathit{Values} & &
\end{array}
$$

Figure 4.19: The final syntax of our Erlang subset. In the last step we added a proper case-operator.

must be written e.g.

$$
\begin{array}{l}
X_1 = E_1, \\
X_2 = E_2, \\
X = f(X_1, X_2), \\
E
\end{array}
$$

This form fixes evaluation order. In Erlang, composite expressions like the first form is allowed and evaluation order is *not* specified.

$$\frac{}{\langle \rho' : \rho \,!\, v, E\rangle \| \mathit{ether}_\rho(v_1\cdots v_n) \to \langle \rho' : E\rangle \| \mathit{ether}_\rho(v_1\cdots v_n v)}\quad \textsf{React1}$$

$$\frac{}{\mathit{ether}_\rho(v_1\cdots v_n) \| \mathit{inbox}_\rho(v'_1\cdots v'_m) \to \mathit{ether}_\rho(v_2\cdots v_n) \| \mathit{inbox}_\rho(v'_1\cdots v'_m v_1)}\quad \textsf{React2}$$

$$\frac{k = \min\{i \mid v_i \in \mathit{matches}(M)\}}{\langle \rho : X = \mathit{receive}\ M, E\rangle \| \mathit{inbox}_\rho(v_1\cdots v_n) \to \langle \rho : \{v_k/X\}E\rangle \| \mathit{inbox}_\rho(v_1\cdots v_{k-1}v_{k+1}\cdots v_n)}\quad \textsf{React3}$$

$$\frac{f(X_1,\ldots,X_n) \overset{\triangle}{=} E_f \qquad \rho' \notin \mathit{fn}(\langle \rho : E\rangle) \qquad P = \langle \rho' : \{v_1/X_1\}\cdots\{v_n/X_n\}E_f\rangle}{\langle \rho : X = \mathit{spawn}(f,[v_1,\ldots,v_n]), E\rangle \to \mathit{new}\ \rho'\ (\langle \rho : \{\rho'/X\}E\rangle \| P \| \mathit{inbox}_{\rho'}() \| \mathit{ether}_{\rho'}() \| \mathit{stack}_{\rho'}())}\quad \textsf{Spawn}$$

$$\frac{}{\langle \rho : X = \mathit{self}(), E\rangle \to \langle \rho : \{\rho/X\}E\rangle}\quad \textsf{Self}$$

$$\frac{(i = 1 \wedge v = v') \vee (i = 2 \wedge v \neq v')}{\langle \rho : \mathit{case}\ v\ \mathit{of}\ v' \to E_1;\ \_ \to E_2\rangle \to \langle \rho : E_i\rangle}\quad \textsf{Test}$$

$$\frac{(v = \mathit{cons}\ v_1\ v_2 \wedge E = \{v_1/X\}\{v_2/X'\}E_1) \vee (\mathit{atom}(v) \wedge E = E_2)}{\langle \rho : \mathit{case}\ v\ \mathit{of}\ [X|X'] \to E_1;\ \_ \to E_2\rangle \to \langle \rho : E\rangle}\quad \textsf{Destruct}$$

$$\frac{f(X_1,\ldots,X_n) \overset{\triangle}{=} E_f \qquad E' = \{v_1/X_1\}\cdots\{v_n/X_n\}E_f}{\langle \rho : X = f(v_1,\ldots,v_n), E\rangle \| \mathit{stack}_\rho(L) \to \langle \rho : E'\rangle \| \mathit{stack}_\rho((X,E)\ L)}\quad \textsf{Call}$$

$$\frac{}{\langle \rho : v\rangle \| \mathit{stack}_\rho((X,E)\ L) \to \langle \rho : \{v/X\}E\rangle \| \mathit{stack}_\rho(L)}\quad \textsf{Return}$$

$$\frac{}{\langle \rho : X = [], E\rangle \to \langle \rho : \{\mathit{nil}/X\}E\rangle}\quad \textsf{Nil}$$

$$\frac{}{\langle \rho : X = [v_1|v_2], E\rangle \to \langle \rho : \{\mathit{cons}\ v_1\ v_2/X\}E\rangle}\quad \textsf{Cons}$$

$$\frac{P \to P''}{P \| P' \to P'' \| P'}\quad \textsf{Par}$$

$$\frac{P \to P'}{\mathit{new}\ \rho\ P \to \mathit{new}\ \rho\ P'}\quad \textsf{Res}$$

$$\frac{P_1 \to P_2 \qquad P_1 \equiv P'_1 \qquad P_2 \equiv P'_2}{P'_1 \to P'_2}\quad \textsf{Struct}$$

Figure 4.20: Final version of the Erlang semantics.

Function declarations and hot code swapping As mentioned, the function definitions are external to our system. If this is not satisfactory, one could use the same mechanism that was used with the inbox and the call stack: let the function definitions float in the tree of processes, and change e.g. the call rule to

$$\frac{E' = \{v_1/X_1\}\cdots\{v_n/X_n\}E_f}{\langle \rho : X = f(v_1,\ldots,v_n),E\rangle \| stack_\rho(L) \| def_{f(X_1,\ldots,X_n)E_f} \to \langle \rho : E'\rangle \| stack_\rho((X,E)\ L) \| def_{f(X_1,\ldots,X_n)E_f}}$$

Of course, the function defintions should then be incorporated into the inital state. This would also allow you to support hot code swapping, e.g.

$$\frac{}{def_{f(X_1,\ldots,X_n)E_f} \| def_{g(X_1,\ldots,X_m)E_g} \| \langle \rho : load(f,g),E\rangle \to def_{f(X_1,\ldots,X_m)E_g} \| def_{g(X_1,\ldots,X_m)E_g} \| \langle \rho : E\rangle}$$

Observations Finally, many process calculi (including π-calculus, [Mil99]) have a semantics where *observations* of communication are more explicit. We believe that our semantics might also benefit from such a formulation.

4.8 Related work

In this section we summarize the two alternative formal definitions of Erlang semantics known to us. We also very briefly compare each one of these to our own. Both papers that we present define the semantics with the aim of facilitating verification of program properties. Beware that although each paper name the Erlang subset that they consider *Core Erlang*, both subsets are much smaller than the more official Core Erlang presented in [CGJ$^+$00].

Toward Parametric Verification of Open Distributed Systems As the title indicates, the paper by Dam, Fredlund and Gurov ([DFG98]) is on the topic of program property verification. Properties are specified in first-order μ-calculus extended with Erlang primitives. The programs considered are from a subset of Erlang which is very similar to our own, except that they allow the name of a called function to be value of an expression (this value must be an atom – there is no lambda abstraction).

The authors define a process p to be a triple $\langle e,\rho,q\rangle$, where e is an expression, ρ is a process ID, and q is a queue of values. The latter is the inbox of the process. They define a *system state* to be a set s of processes such that

$$\langle e,\rho,q\rangle, \langle e',\rho',q'\rangle \in s \wedge \langle e,\rho,q\rangle \neq \langle e',\rho',q'\rangle \Rightarrow \rho \neq \rho'$$

Their semantics is a structured operational semantics, defined by three different relations:

$$\begin{array}{ccc} e & \rightarrow & e' \\ p & \overset{\alpha}{\rightarrow} & p' \\ s & \overset{\alpha}{\rightarrow} & s' \end{array}$$

where e and e' are expressions, p and p' are processes, s and s' are system states, and α is either empty or an action of the form $\rho!v$ or $\rho?v$, where v is a value.

Their semantics of message transmission is equivalent to the one in Figure 4.9; it does not model the fact that messages may be "in the ether". Process IDs are not bound by *new* operator, so all handling of process ID namespaces in the system state is implicit.

The authors do not assume that evaluation order has been settled syntactically (as discussed in Section 4.7), so they handle this issue by generalizing expressions to *reduction contexts* that always expose the left-most reducible subexpression. A further complication is that pattern matching and value deconstruction are kept in their most general form (and not separated), so that the meanings of *case* and *receive* must be defined in terms of *most general unifiers*.

Verification of Erlang Programs using Abstract Interpretation and Model Checking This paper ([Huc99]) by Huch addresses the issue of verifying program properties specified in *Linear Temporal Logic*. Observing that this is not in general decidable for Erlang programs, the author constructs an operational semantics for which the states can be abstracted to a finite domain. The semantics describes an Erlang subset that resembles the one we consider.

Like the above paper, Huch defines a process p to be a triple, here (ρ, e, q), where again ρ is a process ID, e is an expression, and q is a queue of values. A *state* Π is defined to be a finite set of processes.

This paper also uses a structured operational semantics. Unlike the above paper these is only one relation:

$$\Pi \overset{\alpha}{\Rightarrow} \Pi'$$

where α is a label that is either empty or of one of the forms $!v$, $?v$ or $\texttt{spawn}(f)$, where v is a value and f is a function symbol. There is an explicit error state that will be entered if the recipient part of a send expression is not a process ID, or if an unassigned variable is used. In [DFG98], as in our semantics, these situations merely result in a state that cannot be further evaluated.

As in [DFG98], the semantics of message transmission does not model that messages may be “in the ether”, and handling of process ID namespaces in the system state is implicit. Huch also fixes evaluation though the use of reduction contexts and defines the meanings of *case* and *receive* in terms of unification.

4.9 Conclusion

This chapter has developed a formal semantics for Erlang in details. It has also reviewed alternative defintions of such a semantics. Our semantics models the Erlang’s sophisticated message transmission feature and explicitly handles namespaces (of process IDs) through structural congruence as in the π-calculus. The semantics is also unusually modular.

Future work for this topic is to extende the semantics to model hot code swapping: the ability of Erlang to change the definitions of functions at run-time. We would also like to see an extended semantics that supports *observations* of communication (as in the π-calculus), thus allowing standard definitions of simulation and bisimulation.

5 - Towards partial evaluation of Erlang

In this chapter we take some first steps towards the construction of a partial evaluator for the Erlang language. At the present, no one has implemented such a system.

We present a number of motivating examples for partial evaluation (*PE*) of Erlang programs. Section 5.1 starts with a classical example: the power function. The next three sections follow up with examples of truly concurrent programs. We focus on one particular use of partial evaluation in concurrency: to fuse a client and a server at specialization-time, leaving only the client present at run-time. The first example considers a very simple database server and an even simpler client (Section 5.2). The next example suggests the partial evaluation of a more interesting client to the same database server (Section 5.3). Our final example, in Section 5.4, shows a small interpreter that we find fit for partial evaluation. In Section 5.5, we discuss the problem of defining partial evalution in a concurrent setting and propose a reasonably simple such definition. We review related work on partial evaluation of concurrent languages in Section 5.6, and Section 5.7 concludes.

Note that our examples are based on a syntax that is essentially the same as Figure 4.1. Our discussion is based on the semantics of Figure 4.20, although this does not cover the full syntax of our examples.

Acknowledgement The author would like to thank John Launchbury for instigating this work, and for many inspiring discussions on the topic.

5.1 A simple example of partial evaluation

Examples motivating the use of partial evaluation in ordinary sequential or functional programming are legion, see [JGS93]. The most classical among these examples is the power function (see Figure 5.1).

This function demonstrates the advantage of partial evaluation in a simple manner. Specializing the code in the figure wrt. $n = 3$ should give you something like this:

```
exp_d3(X) ->
    X * X * X * 1
end.
```

```
exp(X,N) ->
    case N of
        0 ->
            1;
        _ ->
            X * exp(X,N-1)
    end.
```

Figure 5.1: The power function.

The specialized version is more efficient – but less general – than the original.

Can we state the relationship more precisely? Yes, if we use our semantics for Erlang, we can deduce that for all v, v':

$$\begin{gathered}\langle \rho : X = exp(v,3), E\rangle \| stack_\rho(L) \rightarrow^* \langle \rho : \{v'/X\}E\rangle \| stack_\rho(L) \\ \Longleftrightarrow \\ \langle \rho : X = exp\_d3(v), E\rangle \| stack_\rho(L) \rightarrow^* \langle \rho : \{v'/X\}E\rangle \| stack_\rho(L)\end{gathered}$$

Of course this holds exactly when $v' = v^3$.

This correspondence does not show how the specialized function is more efficient than the original, but we hope that the reader will be able to accept that claim without proof.

5.2 The world's simplest database application

Without the concurrent aspect of Erlang, the implementation of a PE for Erlang could be performed using standard techniques. What we want to show in this and the following sections is examples where specialization in the context of concurrent Erlang programs would be useful.

In Figure 5.2 we show a naive database server. The server is a non-terminating process, the internal state of which is a list of key/value-pairs. Other processes communicate with the server through messages and may either ask for the value associated with a given key or set an entirely new server state.

Now, if we know all the clients to the server, we may be able to eliminate the server by specialization. As an example, consider the case where the only client to the server is the one listed in Figure 5.3.

The client instructs the server to map 1,2 and 3 to their squares, then proceeds to sum the square of 2 with the square of 3. Clearly, the communication between

```
db(State) ->
    receive
        {set,Pid,NewState} ->
            Pid ! ok,
            db(NewState);
        {read,Pid,Key} ->
            Pid ! read(State,Key),
            db(State)
    end.

read(List,Key) ->
    case List of
        [{Key,Value}|_] ->
            Value;
        [_|Rest] ->
            read(Rest,Key)
    end.
```

Figure 5.2: The database server. This process simply has an associative list as its internal state.

```
start_toy() ->
    toy_client(spawn(db,[[]])).

toy_client(Server) ->
    Server ! {set,self(),[{1,1},{2,4},{3,9}]},
    Server ! {read,self(),2},
    Server ! {read,self(),3},
    receive ok -> receive X -> receive Y -> X+Y
    end   end        end.
```

Figure 5.3: A toy database client.

the two processes is not necessary, and partial evaluation applied to this example ought to yield something like:

```
start_toy_spec() ->
    13.
```

Of course this example is somewhat trivial as there is no dynamic data. It was just meant to illustrate how specialization may be able to remove communication and indeed change the process structure (remove a process entirely). A crucial observation is that $\exists v \exists P_{\mathtt{db}}$:

$$\begin{array}{rl} \langle \rho : X = \mathit{start\_toy}(), E\rangle \| \mathit{stack}_{\rho}(L) \| \mathit{inbox}_{\rho}() \| \mathit{ether}_{\rho}() & \rightarrow^{*} \\ \mathit{new}\ \rho'\ P_{\mathtt{db}} \| \langle \rho : \{v/X\}E\rangle \| \mathit{stack}_{\rho}(L) \| \mathit{inbox}_{\rho}() \| \mathit{ether}_{\rho}() & \\ & \\ \mathit{and} & \\ & \\ \langle \rho : X = \mathit{start\_toy\_spec}(), E\rangle \| \mathit{stack}_{\rho}(L) \| \mathit{inbox}_{\rho}() \| \mathit{ether}_{\rho}() & \rightarrow^{*} \\ \langle \rho : \{v/X\}E\rangle \| \mathit{stack}_{\rho}(L) \| \mathit{inbox}_{\rho}() \| \mathit{ether}_{\rho}() & \end{array}$$

The two statements hold when $v = 13$ and $P_{\mathtt{db}}$ represents the blocked remains of the `db` server. We will discuss the significance of this relationship in Section 5.5.

5.3 Adding spice to the database client

Now lets look at a more interesting and – slightly – more realistic situation. In this case the only client to the database server is the one in Figure 5.4.

The client basically interprets a very simple language. Its dynamic input is a list of commands to be executed in sequence. A command may either

- instruct the server to map 1,2,3 to their squares,
- instruct the server to map 1,2,3 to their cubes, or
- look up the value associated with a key.

Interpretation of a list of commands results in the sum of the results of its look-up operations.

A noteworthy feature of this client is that the possible server states are static. A dynamic command list can choose between them, but it cannot instruct the server to go into a state that isn’t already hardcoded in the client. This should allow a partial evaluator to produce a specialized client as shown in Figures 5.5 and 5.6.

```
start(L) ->
    client(spawn(db,[[]]),L).

client(Server,L) ->
    case L of
        [] ->
            0;
        [squares|Rest] ->
            Server ! {set,self(),[{1,1},{2,4},{3,9}]},
            receive
                ok -> client(Server,Rest)
            end;
        [cubes|Rest] ->
            Server ! {set,self(),[{1,1},{2,8},{3,27}]},
            receive
                ok -> client(Server,Rest)
            end;
        [N|Rest] ->
            Server ! {read,self(),N},
            receive
                M -> M + client(Server,Rest)
            end
    end.
```

Figure 5.4: A more complicated database client.

In parallel to the previous example, the fact to observe is that $\forall v \exists v' \exists P_{\mathtt{db}}$:

$$\begin{array}{rl} \langle \rho : X = \mathit{start}(v), E\rangle \| \mathit{stack}_\rho(L) \| \mathit{inbox}_\rho() \| \mathit{ether}_\rho() & \rightarrow^* \\ \mathit{new}\ \rho'\ P_{\mathtt{db}} \| \langle \rho : \{v'/X\}E\rangle \| \mathit{stack}_\rho(L) \| \mathit{inbox}_\rho() \| \mathit{ether}_\rho() & \\ \\ \wedge & \\ \\ \langle \rho : X = \mathit{start\_spec}(v), E\rangle \| \mathit{stack}_\rho(L) \| \mathit{inbox}_\rho() \| \mathit{ether}_\rho() & \rightarrow^* \\ \langle \rho : \{v'/X\}E\rangle \| \mathit{stack}_\rho(L) \| \mathit{inbox}_\rho() \| \mathit{ether}_\rho() & \end{array}$$

(assuming v ranges over values that represent correct command lists). This time, the relationship between v and v' is more complicated. Process $P_{\mathtt{db}}$ is again the blocked remains of the db server.

```
start_spec(L) ->
    client1(L).
client1(L) ->
    case L of
        [] ->
            0;
        [squares|Rest] ->
            client2(Rest);
        [cubes|Rest] ->
            client3(Rest);
        [N|Rest] ->
            read1(N) + client1(Server,Rest)
    end.
client2(L) ->
    case L of
        [] ->
            0;
        [squares|Rest] ->
            client2(Rest);
        [cubes|Rest] ->
            client3(Rest);
        [N|Rest] ->
            read2(N) + client2(Server,Rest)
    end.
client3(L) ->
    case L of
        [] ->
            0;
        [squares|Rest] ->
            client2(Rest);
        [cubes|Rest] ->
            client3(Rest);
        [N|Rest] ->
            read3(N) + client3(Server,Rest)
    end.
```

Figure 5.5: A specialized version of the database client. The specialized read-functions are listed in Figure 5.6.

```
read1(Key) ->
    fail().
read2(Key) ->
    case Key of
        1 -> 1;
        2 -> 4;
        3 -> 9;
        _ -> fail()
    end.
read3(Key) ->
    case Key of
        1 -> 1;
        2 -> 8;
        3 -> 27;
        _ -> fail()
    end.
fail() ->
    case 1 of 0 -> 0 end.
```

Figure 5.6: The specialized read-functions. Function *fail* is provided by the partial evaluator and always causes a pattern match failure.

5.4 An example interpreter

In this section we give a larger motivating example. Our source program (in Figures 5.7 and 5.8) is an interpreter for a small programming language. A program in this language (which is not Turing-complete) simulates a recurrent neural network (i.e. one with cycles) for a number of time steps. Neurons are modeled by processes, and the activation of a neuron (firing) is modeled by messaging. To make the example smaller, there is no concrete syntax for the input programs. The interpreter takes as arguments a list of neurons, a list of initial states of these neurons, the number of time steps to simulate and the name of the neuron whose final state is the output of the program.

An example input network is shown in Figure 5.9. Each node represents a neuron. The number inside the node is the neuron's *activation threshold*: the minimal input that neuron must receive before firing. When a neuron fires (i.e. when the sum of its inputs is above the threshold), it sends input to the neurons that the node has edges to. The amount of input is equal to the number on the given edge.

The `Start` parameter to our interpreter sets the initial input to the neurons in the network. If, for instance, the example network is initated with `Start = [1,1,0]`, nodes A and B would fire in the first time step, but node C would not. In the second time step only node C would fire. In the third step only node A would fire. After this, none of the nodes would fire again.

Given a static network, a static choice of output neuron and a static time limit, we would expect partial evaluation to be able to produce a residual function `int(Start)` consisting of one large case-structure. The residual function represents a compilation of the given program into Erlang.

5.5 Discussion

Modelling the unknown Partial evaluation is usually set in a situation where we have some of the information needed to do a computation. I.e. we can split the needed information into the "known" and the "unknown".

In sequential languages "the unknown" is just dynamic input. This reflects a situation where the context is a system that may call your source program with different values appearing in place of the dynamic parameters. That's the only way external systems can interact with your source code. The static parameters reflect limitations in those interactions.

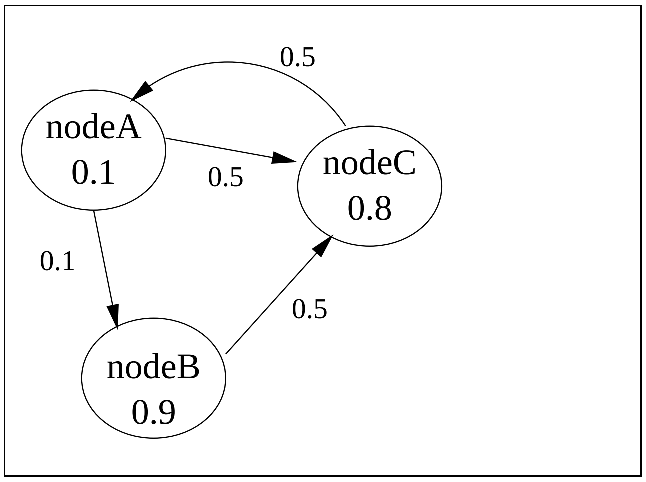
The situation may be different in a concurrent system. We might want the specialized code to interact with *unknown processes*. Instead of just calling a residual function, the environment may want to exchange messages with a process executing the residual code. This is a new kind of "unknown", and it complicates

```
int(Network,Start,Time,Output) ->
    Dictionary = net_spawn(Network),
    broadcast(Dictionary,Dictionary),
    start(Start,Dictionary),
    loop(Time,Dictionary,lookup(Output,Dictionary)).
net_spawn(Network) ->
    case Network of
        [] -> [];
        [{Name,Threshold,Synapses}|Rest] ->
            [{Name,spawn(neuron_init,[Name,Threshold,Synapses])}|
             net_spawn(Rest)]
    end.
broadcast(Dictionary,Msg) ->
    case Dictionary of
        [] -> [];
        [{Name,Pid}|Rest] ->
            Pid ! Msg,
            broadcast(Rest,Msg)
    end.
start(Start,Dictionary) ->
    case Dictionary of
        [] -> [];
        [{Name,Pid}|DictRest] ->
            [Input|StartRest] = Start,
            Pid ! Input,
            start(StartRest,DictRest)
    end.
lookup(Name,Dictionary) ->
    case Dictionary of
        [{Name,Pid}|Rest] -> Pid;
        [Other | Rest] -> lookup(Name,Rest)
    end.
loop(Time,Dictionary,Output) ->
    broadcast(Dictionary,{step,self()}),
    Msg = get_all(Dictionary,Output),
    case Time of
        0 -> Msg;
        N -> loop(N-1,Dictionary,Output)
    end.
```

Figure 5.7: Simulation of a recurrent neural network, page 1.

```
get_all(Dictionary,Output) ->
    case Dictionary of
        [] -> [];
        [Neuron|Rest] ->
            receive
                {Output,N} ->
                    get_all(Rest,Output),
                    N;
                Other -> get_all(Rest,Output)
            end
    end.
neuron_init(Name,Threshold,Synapses) ->
    receive Dictionary ->
            NewSynapses = translate(Synapses,Dictionary),
            neuron(0,Threshold,NewSynapses)
    end.
translate(Neurons,Dictionary) ->
    case Neurons of
        [] -> [];
        [{Name,Wt}|Rest] ->
            [{lookup(Name,Dictionary),Wt} | translate(Rest,Dictionary)]
    end.
neuron(Input,Threshold,Neurons) ->
    receive
        {step,Parent} ->
            case (Input > Threshold) of
                true -> propagate(Neurons),
                        Parent ! {self(),1};
                _ -> propagate([]),
                     Parent ! {self(),0}
            end,
            neuron(0,Threshold,Neurons);
        N -> neuron(Input+N,Threshold,Neurons)
    end.
propagate(Neurons) ->
    case Neurons of
        [] -> [];
        [{Pid,Wt}|Rest] ->
            Pid ! Wt,
            propagate(Rest)
    end.
```

Figure 5.8: Simulation of a recurrent neural network, page 2.



```
[{nodeA,0.1,[{nodeB,0.1},{nodeC,0.5}]},
 {nodeB,0.9,[{nodeC,0.5}]},
 {nodeC,0.8,[{nodeA,0.5}]}
]
```

Figure 5.9: A small recurrent neural network, illustrated graphically and coded as input to our simulator.

specialization. In particular, if unknown processes will have access to the ID of a process that executes residual code, we must residualize all *receive* expressions, as we cannot statically know which messages will be matched by the specified pattern (even if that pattern is static). Unknown processes could dynamically insert values in a process's inbox.

Our examples The above examples focus on the possibility of fusing a client and a server at specialization-time, leaving only the client present at run-time.

The examples were in fact chosen to demonstrate cases where the "unknown" is of the classical, simple kind: in each case, a client function is called and we ask for its return value. In particular, the caller of the client does not send messages to the code being specialized or expect the client to send messages back. Apart from call parameters and return values, all communication performed is internal to the code we want to specialize. And even if the server persists, as the database server in Figure 5.2, all references to it (i.e. all copies of its process ID) are discarded when the client function returns.

This was reflected in our semantical deductions. Recall for instance the observation regarding our non-trivial database client, we observed $\forall v \exists v' \exists P_{\mathtt{db}}$:

$$\begin{array}{r}
\langle \rho : X = start(v), E \rangle \| stack_{\rho}(L) \| inbox_{\rho}() \| ether_{\rho}() \quad \rightarrow^{*} \\
new\ \rho'\ P_{\mathtt{db}} \| \langle \rho : \{v'/X\} E \rangle \| stack_{\rho}(L) \| inbox_{\rho}() \| ether_{\rho}() \qquad \\
\\
\wedge \qquad \\
\\
\langle \rho : X = start\_spec(v), E \rangle \| stack_{\rho}(L) \| inbox_{\rho}() \| ether_{\rho}() \quad \rightarrow^{*} \\
\langle \rho : \{v'/X\} E \rangle \| stack_{\rho}(L) \| inbox_{\rho}() \| ether_{\rho}() \qquad
\end{array}$$

Notice the following:

- We assume the client process's inbox and ether to be empty in both cases. If they were not, the messages in them would interfere with the static client-server communication.
- In the unspecialized version, there can be no external references to the server process. That is, no matter which context you insert the final state into, the process ID ρ' cannot be captured. Only the server itself appears under the binding of ρ'.

The first item has as consequence that we may statically know the contents of the inbox. This allows us to specialize the source code *without* residualizing all *receive* expressions. The second item has as consequence that after specialization, the server does not need to be present at run-time. It is not hard to see that when

$$P = new\ \rho'\ (\langle \rho' : X = receive\, M, E \rangle \| stack_{\rho'}(L) \| inbox_{\rho'}() \| ether_{\rho'}())$$

then P is really blocked in any context, i.e.

$$\begin{array}{c}
P \| P' \rightarrow Q \\
\Leftrightarrow \\
Q \equiv P \| P'' \ \wedge\ P' \rightarrow P''
\end{array}$$

Correctness of Erlang partial evaluation A program transformation like partial evaluation is expected to be meaning-preserving, i.e. not change the meaning of a program during transformation. So before we design partial evaluator we should make more clear *what* the term meaning-preserving amounts to.

In ordinary partial evaluation, the specification of "meaning-preserving" is like this:

$$\forall p, s, d. [\![[\![\mathbf{spec}]\!] (\mathbf{p}, \mathbf{s})]\!] (\mathbf{d}) = [\![\mathbf{p}]\!] (\mathbf{s}, \mathbf{d})$$

where $[\![\cdot]\!]$ defines the (denotational) semantics of programs. For an operational semantics we would get a similiar definition: when **spec** applied to (p, s) yields

p_s we must always have that p_e applied to d evaluates to the same value as p applied to (s,d) does.

For a concurrent program things may be more complicated. Concurrency introduces inherent nondeterminism. Concurrency also means that nontermination plays a different role. In a sequential program nontermination is considered undesirable. In concurrent programs – for example in servers – termination may be very undesirable. The whole point of a server is for it to keep running and answer requests.

We therefore suggest a definition of correct specialization that is inspired by the concept of *simulation* (see e.g. [Mil99]).

Definition 1. Let Γ be a set of function definitions such that f is defined as an N-ary function in Γ. We define

$$\begin{aligned}\mathit{Return}_{f,\Gamma}(v_1,\ldots,v_N) = \{v \mid \exists P:\ & \mathit{InitialState}(\rho, f(v_1,\ldots,v_N) \quad \rightarrow^*_\Gamma \\ & \langle \rho : v\rangle \| \mathit{stack}_\rho() \| P \qquad\qquad \}\end{aligned}$$

Definition 2. Let Γ define f as an $n+m$-ary function and Γ' define f' as an m-ary function. We say that (f',Γ') specializes (f,Γ) with respect to values $v_1,\ldots,v_n$ when the following two properties hold for all $v'_1,\ldots,v'_m$:

$$\mathit{Return}_{f',\Gamma'}(v'_1,\ldots,v'_m) \subseteq \mathit{Return}_{f,\Gamma}(v_1,\ldots,v_n,v'_1,\ldots,v'_m) \tag{5.1}$$

$$\mathit{Return}_{f',\Gamma'}(v'_1,\ldots,v'_m) = \emptyset \Rightarrow \mathit{Return}_{f,\Gamma}(v_1,\ldots,v_n,v'_1,\ldots,v'_m) = \emptyset \tag{5.2}$$

This definition reflects the common simplicity of our examples:

- It does not discern between looping and blocked processes (both result in an empty *Return* set).
- It does not say anything about functions that return new process IDs (this also results in an empty *Return* set).

5.6 Related work on partial evaluation

In this section we briefly review the papers that seem most relevant to the design of an Erlang PE. While no one has constructed a partial evaluator for Erlang, people have considered languages with related issues.

There have also been papers on different optimizations of Erlang, but we have not found any that address matters that are essential to the design of a partial evaluator.

Online PE and its correctness Hosoya et al. ([HKY96]) define an online partial evaluator for a subset of the concurrent language HACL. Their focus is on providing a partial evaluator that is provably correct.

The language is a simple, functional language with a syntax akin to the one we introduced for processes in Section 4.2. HACL processes do not have identifiers; they communicate through named channels as in the π calculus. Recursion is implemented in a standard way and not by means of replication.

The meaning of a process is specified by an operational semantics defining a relation of the form

$$\Gamma \triangleright P \rightarrow \Gamma \triangleright P'$$

where Γ is a set of function definitions and P and P' are processes. There are no actions associated with the relation. Processes are subject to a structural congruence like the one in Figure 4.5.

Partial evaluation is defined by a relation $\rightsquigarrow$ which is also of the form

$$\Gamma \triangleright P \rightsquigarrow \Gamma \triangleright P'$$

A pair $(\Gamma \triangleright P_0)$ can be specialized by the partial evaluator by putting P in a static context $C_s[\cdot]$ and applying $\rightsquigarrow$:

$$\Gamma \triangleright C_s[P_0] \rightsquigarrow^* \Gamma \triangleright P_1$$

The specialization P_1 can then be applied by putting it in a dynamic context and executing the resulting program:

$$\Gamma \triangleright C_d[P_1] \rightsquigarrow \cdots$$

The PE can handle partially static data structures and it removes communication that can be performed at specialization time. This means that it also may remove some nondeterminism in the input program.

It is clear that bisimulation cannot define correctness of a transformation that reduces nondeterminism. Hosoya et al. base their correctness definition on the notion of *barb-agreed quasi-congruence*. A barb of a process P is a channel that the context of P can communicate with P on. The paper discusses the finer issues of choosing a barb-based relation that expresses the right kind of correctness.

PE Techniques for Concurrent Programs Marinescu et al. ([MG97]) present an offline partial evaluator for a variation of CSP (Concurrent Sequential Processes). They prove the correctness of the partial evaluator and discuss a number of practical issues.

The language has a fairly complex syntax. The authors use the term *thread* to denote a sequential unit of execution to avoid confusion with Milner's use of the term *process*. A thread is a sequence of assignments, send operations, receive operations, thread spawns and guarded commands; recursive threads are not allowed. A given program has a fixed number of communication channels. Communication along a channel is synchronous and unidirectional. The semantics is given by an structural operational semantics, defining relation

$$\begin{array}{l} S \xrightarrow{\lambda} S' \\ \textit{where } S, S' \subseteq \textit{COM} \times \textit{ENV} \\ \textit{and } \lambda \in \{\alpha\ \textit{op}\ v \mid \alpha \in \textit{Channels},\ \textit{op} \in \{?, !\}\ v \in \textit{Values}\} \end{array}$$

in 24 rules. The set *COM* contains all sequences of commands, *ENV* contains all environments (mappings from variables to values).

The binding-time analysis analyzes expressions, channels, program points, commands and guarded commands. In a given program, each syntactical instance of one of these entities is classified as either static or dynamic. Specialization is defined using a new relation (defined by an operational semantics) that resembles the semantical relation but also specifies the residual code generated with each step taken.

The correctness definition is quite different from the one in [HKY96]. The authors define a domain of *meanings* of programs, each meaning being an observable trace (ways in which the context can interact with the program) and a possible end-configuration (∞ if the program is nonterminating). Correctness is defined using a relation $\equiv$ on these meanings.

Self-applicable partial evaluation for π calculus Gengler and Martel ([GM96]) define a self-applicable offline partial evaluator for asynchronous π calculus and prove its correctness.

The specialization is performed by an extended self-interpreter in the π calculus. The authors show how to encode π-processes into the π calculus itself and write a self-interpreter `Eval` based on this encoding. The encoding is then extended to two-level processes, and `Eval` is extended appropriately to a partial evaluator `Pev`. When `Pev` is applied to a two-level π-process P, it yields a residual process R which is *weak reduction equivalent* to P (i.e. they are weakly bisimilar wrt. reduction).

The two-level terms that are input to `Pev` are (as is usual) produced by a binding-time analysis. The authors define a relation

$$\Gamma \vdash P : \omega$$

using a simple rule system. P is an source (one-level) process and ω is a two-level annotated version of P. The Γ is called the BTA *assumption*. The assumption is itself a term in the π calculus.

In general terms, $\Gamma \vdash P : \omega$ can be read as: "P may be annotated ω if all static communications in ω are counteracted (reduced by) by Γ". The annotated version ω of a process P presented to Pev should be satisfy

$$\Gamma \vdash P : \omega \wedge \Gamma \rightarrow^{*} \mathbf{0}$$

i.e. its assumption should be weakly bisimilar to the zero process. The BTA assumptions serve to match static communications in different parts of the source process P.

Partial Evaluation of Concurrent Programs Martel and Gengler also described an offline PE for Concurrent ML (in [MG01]). Concurrent ML is like synchronous π calculus without nondeterministic choice and replication but with higher-order functions and recursive definitions.

The specialization is performed by a denotational semantics on two-level terms. Static expressions are reduced, while the semantics of dynamic expressions is defined using external code-building constructors. The two-level expressions are produced by a BTA described in Martel's Ph.d. thesis ([Mar00b]). This BTA uses the result of a control-flow analysis defined in [MG00]. The control-flow analysis "builds a reduced product automaton, which [...] describes an approximation of all possible interleavings of the program" ([MG01], page 506). The paper [MG01] does not discuss correctness criteria but demonstrates some experimental results.

5.7 Conclusion

This chapter has demonstrated how partial evaluation might be useful for the Erlang language. We gave a number of Erlang programs and discussed their specialization. Based on these discussions, we proposed a theoretical definition of correct specialization in Erlang. We also reviewed related work on partial evaluation of other concurrent languages.

There is obviously more work to be done in this area. An actual partial evaluator should be written, and there are still many design issues to settle, e.g. should the system be online of offline, should it handle partially static data structures, and does our focus on client-server fusion limit the usability of such a system too much?

Some of these questions may be answered, if one or more of the related systems reviewed in Section 5.6 can be applied in an *Erlang* setting. But due to the essential differences between concurrency paradigms, such a reapplication will not be not trivial.

6 - Offline Partial Evaluation can be as Accurate as Online Partial Evaluation

In this chapter we show that the accuracy of online partial evaluation, or polyvariant specialization based on constant propagation, can be simulated by offline partial evaluation using a maximally polyvariant binding-time analysis. We point out that, while their accuracy is the same, online partial evaluation offers better opportunities for powerful generalization strategies. Our results are presented using a flowchart language with recursive procedures.

6.1 Introduction

Partial evaluation [JGS93] is a well-known technique for program specialization based on aggressive constant propagation and extensive call unfolding. Strategies for partial evaluation can be categorized as *online* or *offline* depending on whether they take reduction decisions during or before the transformation. The latter utilize a static analysis called *binding-time analysis* to take reduction decisions (BTA) [JSS85]. The benefits of online versus offline partial evaluation have been discussed extensively.

The first partial evaluators were online (*e.g.,* [Fut71, CL73, BHOS76, Ers77]). The offline strategy was originally introduced [JSS85, JSS89] to overcome difficulties associated with self-application of partial evaluators. The offline strategy has proven to be very effective in handling complex language features and combines well with other program analyses (such as pointer analysis [And93, GMS99]). Partial evaluation systems for languages like C [And94, CHN$^+$96], Fortran [KKZG95] and Java [MY01, Sch01] have been implemented based on offline techniques. When comparing these two approaches, technical and pragmatic issues can be considered, such as efficiency of the transformation and ease of use. We disregard pragmatic aspects of partial evaluation and focus entirely on the accuracy of the transformation.

We make use of the clear and instructive presentation of partial evaluation [Hat99] and extend the flowchart language with recursive procedures. Flowchart languages are a classical means to study the essence of partial evaluation (*e.g.*, [Bul84, Jon88, GJ91, Hat99]). They are small enough to allow a clean semantics presentation, while being rich enough to examine a multitude of issues that occur when specializing more realistic languages.

Beside partial evaluation [JGS93], which is based on *constant propagation*, there exist other, powerful specialization techniques based on *unification*, such as supercompilation [Tur86] and partial deduction [LS91], or *theorem proving*, such as generalized partial computation [FN88] and the specialization method in [CL73], which we will not consider in this paper (for a detailed comparison see [GK93, GS94, GS96, SGJ96]).

6.1.1 Anecdotal Evidence

Anecdotal evidence is usually given in the literature discussing the accuracy of partial evaluation [CD91b, Mey91, WCRS91]. Results produced by one system are compared with those produced by another system. Such comparisons are often influenced by subjective design choices and do not always exhibit fundamental differences in the techniques.

The following example is akin to the ones frequently used to illustrate the benefits of online partial evaluation. The program fragment consists of a conditional `case` which, depending on the outcome of the test `s > 0`, passes control either to the block labeled `sets` or `setd`. In these two blocks, variable `x` is updated using the value of `s` or `d`, respectively. In both cases, execution resumes at block `cont`. This example shows a typical control flow in an imperative language.

```
      CASE (s > 0) sets setd
sets: x := s + 1
      GOTO cont
setd: x := d + 2
      GOTO cont
cont: PRINT (x + 3)
```

Let `d` be dynamic. Given static value `s=1`, an online partial evaluator typically reduces this program fragment to one statement:

```
PRINT (5)
```

The static test `s > 0` was decided, and the output value `5` was computed. Most offline partial evaluators (*e.g.*, [JSS89, BD91, And94, HN99]) cannot handle this situation and will typically produce

```
x := 2
PRINT (x + 3)
```

Even though the test was decided, the output value was not computed. This is due to the *monovariant* (uniform, pointwise) BTA used in these offline partial evaluators.[1] The analysis takes a safe assumption regarding the availability of the resulting value. Indeed, this value can be either static or dynamic depending on the value of `s` which is not available to the BTA.

Clearly, the online partial evaluator performs more static computations (reduction to `5`) because it can take advantage of the fact that `s=1` at specialization time, while the offline partial evaluator has made a safe approximation at program point `cont` beforehand and residualized the results of both arms. Knowing the value of `x`, the online partial evaluation may perform further reductions in the program, while the offline partial evaluator will not be able to take advantage of this situation.

But this is *not* the case with the offline partial evaluator defined in this paper. A polyvariant BTA will allow the possibility of *not* residualizing the continuation, and thus reduce the above to one statement as the online system does.

6.1.2 Contribution

The purpose of this paper is to put the comparison of online and offline accuracy on a solid theoretical basis by examining the functional (lack of) difference between online and offline partial evaluation.

As this paper investigates the limits of accuracy in partial evaluation, we introduce an online partial evaluator that has *no intentional loss of precision* and thus is as accurate as possible. We then show that an offline strategy, together with a *maximally polyvariant* BTA [CGL00], has the same accuracy in finding static information as our online system. This is contrary to the common belief that a chief disadvantage of offline strategies is the imprecision caused by the BTA which guides the offline specialization process. The theoretical benefit of an online strategy is therefore *not* that it is more precise than an offline strategy, but that it can make better decisions regarding when to generalize its state, i.e., when to be *imprecise*.

It is crucial to our results that the BTA used is *maximally polyvariant*. This is in contrast to the uniform divisions used by most offline systems:

> ... the same division is valid at all program points. Call such a division uniform. This assumption, which simplifies matters without being essential, often holds in practice although counterexamples may easily be found (...) [JGS93, page 77]

[1]The granularity of BTA is discussed in [JGS93, Section 4.9]; see also [Con93, HDL98, CGL00].

Many systems in practice work well using monovariant BTA. But our results show that choosing monovariance has serious implications for the precision of the analysis, indicating that the designer of an offline partial evaluator should not choose to implement a BTA without considerations. This supports the earlier thesis [Con93, CGL00] that the precision of the BTA should be *parameterized* rather than hard-coded at implementation time (see also the design regrets in [Mar00a]). An example is the parametrization that was done in [CGL00].

It is well-known that the more information a program transformer propagates, the more it will suffer from code explosion and non-termination, and the greater the need will be for a *controlled* loss of information. Our maximally polyvariant BTA is not useful in PE practice without external analyses to handle these issues. A discussion of termination issues in partial evaluation can be found in [JG02]. Similarly, program specialization techniques capturing more information, such as techniques utilizing unification or theorem proving, require more sophisticated termination strategies.

6.1.3 Outline

In Section 6.2, we define a flowchart language. Section 6.3 lays out the framework for defining partial evaluation of that language. In Sections 6.4 and 6.5, we present two partial evaluation systems. Section 6.6 describes block specialization (code generation). Our main technical result – the proof that these two partial evaluators are functionally equivalent – is given in Section 6.7. This is followed by an example illustrating our result. In Section 6.9, we discuss generalization and what offline systems essentially cannot do. Section 6.10 discusses related work and Section 6.11 concludes.

We assume the reader is familiar with the basic notions of partial evaluation (a good source is [Hat99] or [JGS93, Part II]).

6.2 A Flowchart Language with Procedures

This section presents the syntax and semantics of a simple flowchart language. The basic aspects of computation in a flowchart language are store transformations by assignments and control transfers by jumps.

6.2.1 Syntax

Figure 6.1 defines the syntax of a flowchart language, F, with assignments, jumps, and recursive procedures. An F-program is a sequence of labelled basic blocks. A *basic block* contains either an assignment and a jump, a procedure call, or a

Grammar

$$
\begin{array}{lcl}
p & ::= & b^+ \\
b & ::= & l\,\mathbf{:}\ a\ j \\
 & | & l\,\mathbf{:}\ \mathbf{call}\ l\ l \\
 & | & l\,\mathbf{:}\ \mathbf{return}\ x\ l \\
a & ::= & x\ \mathbf{:=}\ o(x^*) \\
j & ::= & \mathbf{case}\ t(x^*)\ l^+
\end{array}
$$

Syntax Domains

p	$\in$	Programs	j	$\in$	Jumps	o	$\in$	Operators
b	$\in$	Basic-Blocks	l	$\in$	Labels	t	$\in$	Tests
a	$\in$	Assignments	x	$\in$	Variables			

Figure 6.1: Syntax of the flowchart language with procedures

return statement. We denote by Basic-Blocks$[p]$ the set of labeled basic blocks in program p. An example program is shown in Figure 6.2.

An *assignment* x **:=** $o(x^*)$ contains the application of an n-ary operator o to arguments x^*. For simplicity, we allow only variables as arguments. Nested expressions can be built using several assignments. Constants are represented by nullary operators. A *jump* **case** $t(x^*)\ l^+$ contains the application of an n-ary test t to arguments x^* and labels l^+. Control will be transferred to one of the labels in l^+.

A *procedure call* **call** $l'\ l''$ transfers control to a procedure whose entry block is labeled l' and copies the whole store. When the procedure returns, control is transferred to label l'' (possibly offset by the label in the return statement; see below) and the value of one specified variable is copied back into the original store. We do *not* demand that there necessarily be one distinguished entry point of a procedure; one may call to any basic block, from which execution will then proceed until a return statement is met.

A *return statement* **return** $x\ l$ returns the value of variable x. The return statement also includes an offset label l which is "added" to the return label given in the procedure call (for simplicity, this offset is also a label). The offset makes it easy to return to different basic blocks in the caller. This will be exploited when we generate residual programs. The concatenation of return label and offset is defined in Section 6.2.2. The special label nil $\in$ Labels is used as the empty offset. We assume that these concatenated target labels all exist in the program. Because of this, and because the number of labels in a program is finite, there will only be finitely many different labels put on the call stack.

To denote syntactic objects of a given F-program p, we write Variables$[p]$, Basic-Blocks$[p]$, *etc.*

```
loop: a := a - i
      CASE (i = 0) done test
test: i := i - 1
      CASE (a = 1) setz loop
setz: z := i
      GOTO loop
done: z := z + 1
      GOTO retn
retn: RETURN z end
```

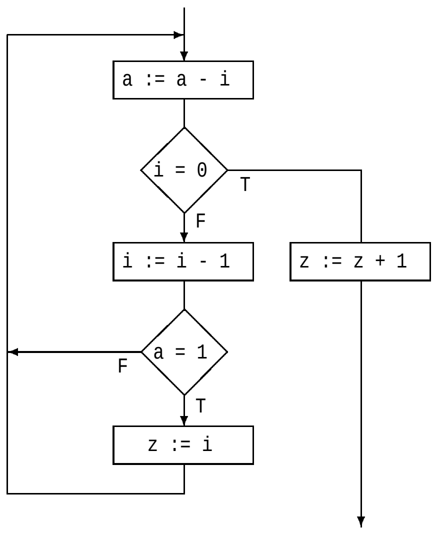


Figure 6.2: Example program *monitor-loop* (with syntactic sugar)

A program p is represented by a *block map* Γ that maps label l into the basic block labeled l in p. We often refer to a program with block map Γ simply as program Γ, and to a basic block labeled l as block l. The first basic block in a program carries the initial label. We assume that every F-program p we consider is well-formed. The definition demands an independence of assignment and jump inside a basic block. It is not hard to transform any program to one that does not violate this demand. This constraint simplifies the semantics definitions.

Definition 3. An F-program p is *well-formed* if,
in every block of p containing an assignment and a jump, the assigned variable x is not part of the condition of the jump.

Example programs in this paper are written using syntactic sugar. We use infix notation for binary operators, and write **goto** l instead of **case** $t_{\texttt{true}}()\ l$.

Example 1. *Program* monitor-loop *(Figure 6.2) contains one loop controlled by induction variable* `i`*. In each iteration, variable* `a` *is decremented by* `i` *and the new value is tested. If* `a` *equals* `1`*, then the current value of* `i` *is saved in* `z`*. When the loop terminates,* `z` *is incremented by* `1`*. The program has no particular purpose.*

6.2.2 Semantics

The evaluation of a program proceeds sequentially from one block to another.

Assignment

$$\frac{\forall i\,.\,\sigma(x_i) = v_i \quad [\![o]\!](v_1 \ldots v_n) = v}{\sigma \vdash x := o(x_1 \ldots x_n) \Rightarrow \sigma[x \mapsto v]}$$

Jump

$$\frac{\forall i\,.\,\sigma(x_i) = v_i \quad [\![t]\!](v_1 \ldots v_n, l_1 \ldots l_m) = l}{\sigma \vdash \textbf{case}\; t(x_1 \ldots x_n)\; l_1 \ldots l_m \Rightarrow l}$$

Basic block

$$\frac{\Gamma(l) = a\; j \quad \sigma \vdash a \Rightarrow \sigma' \quad \sigma \vdash j \Rightarrow l'}{\Vdash_\Gamma (l,\sigma):r \rightarrow_{int} (l',\sigma'):r}$$

$$\frac{\Gamma(l) = \textbf{call}\; l'\; l''}{\Vdash_\Gamma (l,\sigma):r \rightarrow_{int} (l',\sigma):(l'',\sigma):r}$$

$$\frac{\Gamma(l) = \textbf{return}\; x\; l'' \quad \sigma'' = \sigma'[x \mapsto \sigma(x)]}{\Vdash_\Gamma (l,\sigma):(l',\sigma'):r \rightarrow_{int} (l' \cdot l'',\sigma''):r}$$

Semantic Values

r	$\in$	Stacks	$=$	(Labels $\times$ Stores)*
l	$\in$	Labels	$=$	Block-Labels $\cup$ $\{\mathsf{nil}\}$
σ	$\in$	Stores	$=$	Variables $\rightharpoonup$ Values
Γ	$\in$	Block-Maps	$=$	Block-Labels $\rightharpoonup$ Basic-Blocks

Figure 6.3: Operational semantics of flowchart language with procedures

A *computational state* $s \in$ States represents the current state of execution of a procedure. A state is a pair (l,σ) that contains label l of the current basic block and the values of the program's variables in store σ. A *stack of states* $r \in$ Stacks represents the current state of execution of a program.

At a procedure call, the current store is paired with the return label, pushed on the call stack, and control is shifted to the called label. On return, we observe the return offset label l' and the value v of the return variable x. The state (l,σ) is popped off the call stack and the new state will be $(l \cdot l', \sigma[x \mapsto v])$.

Primitives

The semantics is defined *wrt* the semantics of Operators and Tests.

We access the semantics of these by means of an operator $[\![\cdot]\!]$ that works on both operator names and test names. Each operator name is mapped to a function from value tuples to values e.g.:

$$[\![o]\!](v_1 \ldots v_n) = v$$

Each test name is mapped to a function from value/label tuples to labels e.g.:

$$[\![t]\!](v_1 \dots v_n, l_1 \dots l_m) = l$$

Of course, we assume that the test always returns one of its input labels, $l \in \{l_1, \dots, l_m\}$, and that the test does not depend on those labels.[2]

Memory and Control

A *store* $\sigma \in$ Stores is a partial function from Variables to Values. We write $\sigma[x \mapsto v]$ to denote the store that is just like σ except that variable x maps to value v.[3] By $\sigma|_x$ we denote the store that is just like σ except that variable x is not in the domain of σ. We regard labels as strings, and use an auxiliary function, $\cdot$, to concatenate labels (*e.g.*, `cont`$\cdot$`abc` = `contabc`). The empty label, nil, in a return-statement is syntactic sugar for the empty string, ε.

Evaluation

The rules in Figure 6.3 define a transition relation $\rightarrow_{int}$ between stacks of states. A judgement $\Vdash_\Gamma r \rightarrow_{int} r'$ represents a transition from a stack r to a stack r' in a program Γ. We omit Γ when it is obvious from the context, and write $\rightarrow_{int}$ in infix notation. The transitive closure is denoted by $r \rightarrow^*_{int} r'$. The rules are straightforward and should not need particular explanation.

Definition 4. Let p be a program represented by a block map Γ, let σ_0 be an initial store, and let l_0 be an initial label for p, then *program evaluation* $[\![\;]\!]$ is defined as follows:

$$[\![p]\!](l_0, \sigma_0) \overset{\text{def}}{=} \begin{cases} \sigma_0[x \mapsto \sigma(x)] & \text{if } (l_0, \sigma_0) : \varepsilon \rightarrow^*_{int} (l, \sigma) : \varepsilon \wedge \Gamma(l) = \textbf{return}\ x\ l' \\ \text{undefined} & \textit{otherwise.} \end{cases}$$

6.3 Partial Evaluation

From an abstract view, a partial evaluator performs a three-step process [Jon88, Hat99]:

1. Collection of all reachable configurations.
2. Block specialization and code generation.
3. Post-processing (*e.g.*, transition compression).

[2] I.e. if $[\![t]\!](v_1 \dots v_n, l_1 \dots l_m) = l_i$ then $[\![t]\!](v_1 \dots v_n, l'_1 \dots l'_m) = l'_i$ for any choice of $l'_1 \dots l'_m$.

[3] We use this notation even if x was already defined in σ.

The first step determines the set of reachable configurations given an initial configuration. The second step incorporates the values of the known variables into the program points and generates specialized blocks. The third step eliminates redundant code, *e.g.,* jumps caused by blocks chained by one-way jumps. This last step concerns post-processing a residual program. It is identical for online and offline partial evaluation and we will not consider it further.

We focus on the first two steps of which the collection of reachable configurations is the most important step during partial evaluation, while block specialization is straightforward once the set of reachable configurations is computed. In practice, both steps are often carried out in a single phase.

6.3.1 Collecting Reachable Configurations

Configurations

The essential difference between interpretation and partial evaluation is that a partial evaluator operates over a partial store in which not all values of the program variables are known. To keep track of known ($\underline{s}$tatic) and unknown ($\underline{d}$ynamic) values, we use a separate *binding-time store* (bt-store) β which maps variables to static and dynamic tags (S, D), telling us which values are available in the store and which are not. A variable is either static or dynamic; there is no partially static data. A bt-store is also called *division* [Jon88] since it divides the set of program variables into two categories: Static and dynamic.

Formally, a *partial evaluation store* (pe-store) is a pair (β,σ) of bt-store β and store σ such that

- If a variable x has a static value v, then $\beta(x) = S$ and $\sigma(x) = v$.
- If a variable x has a dynamic value, then $\beta(x) = D$ and $\sigma(x)$ is undefined.

In the following sections, we shall introduce relations $\rightarrow_{on}$ and $\rightarrow_{off}$. Rather than relating stacks of states (like $\rightarrow_{int}$), these will relate stacks of configurations. A *configuration* $(l,(\beta,\sigma))$ consists of a label l and a pe-store (β,σ). A configuration represents a set of states that contains all states in which static variables are fixed to their static values.

Reachability

Given an initial configuration stack, the first step of partial evaluation is to compute the set of all *reachable configurations*, called *Poly*. Intuitively, to compute *Poly*, we have to follow all possible computation traces through a program. Let l_0 be the initial label and let (β_0,σ_0) be the initial pe-store. The initial stack is

$(l_0,(\beta_0,\sigma_0)):\varepsilon$ and we consider a configuration $(l,(\beta,\sigma))$ reachable if for some r we have $(l_0,(\beta_0,\sigma_0)):\varepsilon \rightarrow^*_{on} (l,(\beta,\sigma)):r$ (in the offline case substitute $\rightarrow_{on}$ by $\rightarrow_{off}$). *Poly* is the set of top frames of the stacks in the closure. For online partial evaluation, we define $Poly_{on}$, and for offline partial evaluation $Poly_{off}$. These two sets are central to our comparison.

Definition 5.

$$Poly_{on} = \{\,(l,(\beta,\sigma)) \mid (l_0,(\beta_0,\sigma_0)):\varepsilon \rightarrow^*_{on} (l,(\beta,\sigma)):r\,\}$$

$$Poly_{off} = \{\,(l,(\beta,\sigma)) \mid (l_0,(\beta_0,\sigma_0)):\varepsilon \rightarrow^*_{off} (l,(\beta,\sigma)):r\,\}$$

The order in which configurations are collected in *Poly* is not essential for our discussion (depth-first, breadth-first). Also, *Poly* may or may not be infinite. A partial evaluator terminates if *Poly* is finite. This is the case when the static values only vary finitely during all computations in which the static input is fixed and the dynamic input varies freely. In most of the partial evaluation literature, it is considered acceptable for a partial evaluator to loop on some initial configurations. The two partial evaluators defined in this paper are no exception.

In Section 6.7, we prove that our partial evaluators are functionally equivalent: The residual programs they generate are identical. This means that the binding-time analysis of our offline partial evaluator achieves the same accuracy as our online partial evaluator.

For any program Γ and any call stacks r,r' the following holds:

$$\vdash_\Gamma r \rightarrow_{on} r' \iff \vdash_\Gamma r \rightarrow_{off} r'$$

6.3.2 Block Specialization

Block specialization produces the *residual* program that is the result of the specialization process. Each basic block in the residual program corresponds to some basic block of the source program. We construct the residual program block by customizing the source program block *wrt* a set of static values. The block specialization builds directly on top of the collection phase: for each configuration $(l,(\beta,\sigma)) \in Poly$ we must specialize the block labelled l with the static values in σ.

In our system, block specialization is orthogonal to the use of an online or offline strategy for computing *Poly*. This lets us separate the construction of *Poly* from block specialization, thus greatly simplifying our comparison.

Assignment

$$\frac{\forall i\,.\,\beta(x_i) = S \quad \forall i\,.\,\sigma(x_i) = v_i \quad [\![o]\!](v_1 ... v_n) = v}{(\beta,\sigma) \vdash_{on} x \mathrel{\mathbf{:=}} o(x_1 ... x_n) \Rightarrow (\beta[x \mapsto S], \sigma[x \mapsto v])}$$

$$\frac{\exists i\,.\,\beta(x_i) = D}{(\beta,\sigma) \vdash_{on} x \mathrel{\mathbf{:=}} o(x_1 ... x_n) \Rightarrow (\beta[x \mapsto D], \sigma|_x)}$$

Jump

$$\frac{\forall i\,.\,\beta(x_i) = S \quad \forall i\,.\,\sigma(x_i) = v_i \quad [\![t]\!](v_1 ... v_n, l_1 ... l_m) = l}{(\beta,\sigma) \vdash_{on} \mathbf{case}\; t(x_1 ... x_n)\; l_1 ... l_m \Rightarrow l}$$

$$\frac{\exists i\,.\,\beta(x_i) = D \quad l \in \{l_1, \ldots, l_m\}}{(\beta,\sigma) \vdash_{on} \mathbf{case}\; t(x_1 ... x_n)\; l_1 ... l_m \Rightarrow l}$$

Basic block

$$\frac{\Gamma(l) = a\; j \quad (\beta,\sigma) \vdash_{on} a \Rightarrow (\beta',\sigma') \quad (\beta,\sigma) \vdash_{on} j \Rightarrow l'}{\vdash_\Gamma (l,(\beta,\sigma)) : r \rightarrow_{on} (l',(\beta',\sigma')) : r}$$

$$\frac{\Gamma(l) = \mathbf{call}\; l'\; l''}{\vdash_\Gamma (l,(\beta,\sigma)) : r \rightarrow_{on} (l',(\beta,\sigma)) : (l'',(\beta,\sigma)) : r}$$

$$\frac{\Gamma(l) = \mathbf{return}\; x\; l'' \quad \beta'' = \beta'[x \mapsto \beta(x)] \quad \sigma'' = \begin{cases} \sigma'|_x & \text{if } \beta(x) = D \\ \sigma'[x \mapsto \sigma(x)] & \text{if } \beta(x) = S \end{cases}}{\vdash_\Gamma (l,(\beta,\sigma)) : (l',(\beta',\sigma')) : r \rightarrow_{on} (l' \cdot l'',(\beta'',\sigma'')) : r}$$

Semantic Values

r	$\in$	Stacks	$=$	(Labels $\times$ (Bt-Stores $\times$ Stores))*
l	$\in$	Labels	$=$	Block-Labels $\cup$ {nil}
β	$\in$	Bt-Stores	$=$	Variables $\rightharpoonup$ Bt-Values
σ	$\in$	Stores	$=$	Variables $\rightharpoonup$ Values
Γ	$\in$	Block-Maps	$=$	Block-Labels $\rightharpoonup$ Basic-Blocks

Figure 6.4: Online specialization phase: relation $\rightarrow_{on}$

6.4 Online Collection Phase

We will now formalize the online collection phase using an operational semantics that closely follows the structure of the interpreter (Figure 6.3).

The evaluation of operators (and tests) now depends on the bt-store β. If all arguments of an operator are static, the operator is evaluated and yields a value; otherwise, the operator cannot be evaluated and the result becomes undefined. The task of distinguishing the constructs to be evaluated from those that cannot be evaluated is performed *online* as the collection is carried out. In offline partial evaluation, this task is carried out *offline* in a separate phase before collection.

Figure 6.4 defines a transition relation $\rightarrow_{on}$ between stacks of configurations. A judgement $\vdash_\Gamma r \rightarrow_{on} r'$ represents a transition from a stack r to a stack r' in a program Γ. The rules for handling assignments and jumps are simple.

The rule for assignment-jump basic blocks selects an arbitrary label among the labels returned by the jump. This enables the control to be transferred to any label when the test in a jump is dynamic. This leads to the situation of branching traces in partial evaluation (all possible control flows have to be considered). Note that the rule assumes that all programs are well-formed, as described in Definition 3.

The rules for call and return mimic the rules of the original semantics, only in this case, a return must transfer a bt-value and update a partial store rather than just copying a single value.

Example 2. *Consider the program 'monitor-loop' from Figure 6.2 and an initial pe-store where* `i=2`*,* `a=4`*, and* `z` *is dynamic. The set of configurations computed by online partial evaluation is*

$$\begin{aligned}
Poly_{on} = \{\ & (\mathtt{loop},([\mathtt{i}\mapsto S,\mathtt{a}\mapsto S,\mathtt{z}\mapsto D],[\mathtt{i}\mapsto 2,\mathtt{a}\mapsto 4])),\\
& (\mathtt{test},([\mathtt{i}\mapsto S,\mathtt{a}\mapsto S,\mathtt{z}\mapsto D],[\mathtt{i}\mapsto 2,\mathtt{a}\mapsto 2])),\\
& (\mathtt{loop},([\mathtt{i}\mapsto S,\mathtt{a}\mapsto S,\mathtt{z}\mapsto D],[\mathtt{i}\mapsto 1,\mathtt{a}\mapsto 2])),\\
& (\mathtt{test},([\mathtt{i}\mapsto S,\mathtt{a}\mapsto S,\mathtt{z}\mapsto D],[\mathtt{i}\mapsto 1,\mathtt{a}\mapsto 1])),\\
& (\mathtt{setz},([\mathtt{i}\mapsto S,\mathtt{a}\mapsto S,\mathtt{z}\mapsto D],[\mathtt{i}\mapsto 0,\mathtt{a}\mapsto 1])),\\
& (\mathtt{loop},([\mathtt{i}\mapsto S,\mathtt{a}\mapsto S,\mathtt{z}\mapsto S],[\mathtt{i}\mapsto 0,\mathtt{a}\mapsto 1,\mathtt{z}\mapsto 0])),\\
& (\mathtt{done},([\mathtt{i}\mapsto S,\mathtt{a}\mapsto S,\mathtt{z}\mapsto S],[\mathtt{i}\mapsto 0,\mathtt{a}\mapsto 1,\mathtt{z}\mapsto 0])),\\
& (\mathtt{retn},([\mathtt{i}\mapsto S,\mathtt{a}\mapsto S,\mathtt{z}\mapsto S],[\mathtt{i}\mapsto 0,\mathtt{a}\mapsto 1,\mathtt{z}\mapsto 1]))\ \}
\end{aligned}$$

6.5 Offline Collection Phase

In offline partial evaluation, the task of distinguishing constructs to be evaluated from those that cannot be evaluated at specialization time is carried out in a separate step, called binding-time analysis (BTA), before the offline collection phase. Based on this information, the offline collection phase determines whether a construct is to be evaluated or not. The result of BTA is traditionally returned in the form of an annotated version of the source program. We choose to return it simply as a finite relation on stacks (see below).

Figure 6.5 defines a transition relation $\rightarrow_{off}$ between stacks of configurations. A judgement $\vdash_\Gamma r \rightarrow_{off} r'$ represents a transition from a stack r to a stack r' in a program Γ. The rules closely follow the corresponding rules in the online collection phase; indeed the two rules for jumps are identical to the jump rules in the online collection phase. The difference is that only store σ is updated; there is no calculation of a new bt-store. Instead, the first block rule makes use of the bt-relation $\rightarrow_{bta}$ to determine the new bt-store β'. This relation uses stacks of label/bt-store pairs. Thus, we use the following notation.

Assignment

$$\frac{\forall i\,.\,\beta(x_i)=S \quad \forall i\,.\,\sigma(x_i)=v_i \quad [\![o]\!](v_1\ldots v_n)=v}{(\beta,\sigma)\vdash_{\mathit{off}} x \mathrel{:=} o(x_1\ldots x_n) \Rightarrow \sigma[x\mapsto v]}$$

$$\frac{\exists i\,.\,\beta(x_i)=D}{(\beta,\sigma)\vdash_{\mathit{off}} x \mathrel{:=} o(x_1\ldots x_n) \Rightarrow \sigma|_x}$$

Jump

$$\frac{\forall i\,.\,\beta(x_i)=S \quad \forall i\,.\,\sigma(x_i)=v_i \quad [\![t]\!](v_1\ldots v_n, l_1\ldots l_m)=l}{(\beta,\sigma)\vdash_{\mathit{off}} \textbf{case}\; t(x_1\ldots x_n)\; l_1\ldots l_m \Rightarrow l}$$

$$\frac{\exists i\,.\,\beta(x_i)=D \quad l\in\{l_1,\ldots,l_m\}}{(\beta,\sigma)\vdash_{\mathit{off}} \textbf{case}\; t(x_1\ldots x_n)\; l_1\ldots l_m \Rightarrow l}$$

Basic block

$$\frac{\Gamma(l)=a\; j \quad (\beta,\sigma)\vdash_{\mathit{off}} a\Rightarrow\sigma' \quad (\beta,\sigma)\vdash_{\mathit{off}} j\Rightarrow l' \quad (l,\beta):\overline{r}\rightharpoonup_{\mathit{bta}} (l',\beta'):\overline{r}}{\vdash_\Gamma (l,(\beta,\sigma)):r \rightharpoonup_{\mathit{off}} (l',(\beta',\sigma')):r}$$

$$\frac{\Gamma(l)=\textbf{call}\; l'\; l''}{\vdash_\Gamma (l,(\beta,\sigma)):r \rightharpoonup_{\mathit{off}} (l',(\beta,\sigma)):(l'',(\beta,\sigma)):r}$$

$$\frac{\Gamma(l)=\textbf{return}\; x\; l'' \quad \vdash_\Gamma (l,\beta):(l',\beta'):\overline{r}\rightharpoonup_{\mathit{bta}} (l'\cdot l'',\beta''):\overline{r} \quad \sigma''=\begin{cases}\sigma'|_x & \text{if } \beta(x)=D\\ \sigma'[x\mapsto\sigma(x)] & \text{if } \beta(x)=S\end{cases}}{\vdash_\Gamma (l,(\beta,\sigma)):(l',(\beta',\sigma')):r \rightharpoonup_{\mathit{off}} (l'\cdot l'',(\beta'',\sigma'')):r}$$

Semantic Values

r	$\in$	Stacks	$=$	(Labels $\times$ (Bt-Stores $\times$ Stores))*
l	$\in$	Labels	$=$	Block-Labels $\cup$ {nil}
β	$\in$	Bt-Stores	$=$	Variables $\rightharpoonup$ Bt-Values
σ	$\in$	Stores	$=$	Variables $\rightharpoonup$ Values
Γ	$\in$	Block-Maps	$=$	Block-Labels $\rightharpoonup$ Basic-Blocks

Figure 6.5: Offline collection phase: relation $\rightharpoonup_{\mathit{off}}$

Definition 6. The “binding-time” version of a call stack is obtained using the operator $\overline{\bullet}$: (Labels $\times$ (Bt-Stores $\times$ Stores))* $\to$ (Labels $\times$ Bt-Stores)* that drops the store component from a stack recursively by these two equations:

$$\overline{\varepsilon} \;=\; \varepsilon \quad \textit{and} \quad \overline{(l,(\beta,\sigma)):r} \;=\; (l,\beta):\overline{r}$$

Note that the relation $\rightharpoonup_{\mathit{off}}$ is fixed to the maximally polyvariant BTA defined below in Section 6.5.1. Our results only hold for this particular BTA.

Assignment

$$\frac{\forall i\,.\,\beta(x_i) = S}{\beta \vdash_{bta} x := o(x_1 \ldots x_n) \Rightarrow \beta[x \mapsto S]}$$

$$\frac{\exists i\,.\,\beta(x_i) = D}{\beta \vdash_{bta} x := o(x_1 \ldots x_n) \Rightarrow \beta[x \mapsto D]}$$

Jump

$$\frac{l \in \{l_1, \ldots, l_m\}}{\beta \vdash_{bta} \textbf{case}\; t(x_1 \ldots x_n)\; l_1 \ldots l_m \Rightarrow l}$$

Basic block

$$\frac{\Gamma(l) = a\; j \quad \beta \vdash_{bta} a \Rightarrow \beta' \quad \beta \vdash_{bta} j \Rightarrow l'}{\vdash_\Gamma (l,\beta):r \rightarrow_{bta} (l',\beta'):r}$$

$$\frac{\Gamma(l) = \textbf{call}\; l'\; l''}{\vdash_\Gamma (l,\beta):r \rightarrow_{bta} (l',\beta):(l'',\beta):r}$$

$$\frac{\Gamma(l) = \textbf{return}\; x\; l'' \quad \beta'' = \beta'[x \mapsto \beta(x)]}{\vdash_\Gamma (l,\beta):(l',\beta'):r \rightarrow_{bta} (l' \cdot l'',\beta''):r}$$

Semantic Values

$$\begin{array}{llll} r & \in & \text{Stacks} & = & (\text{Labels} \times \text{Bt-Stores})^* \\ l & \in & \text{Labels} & = & \text{Block-Labels} \cup \{\mathsf{nil}\} \\ \beta & \in & \text{Bt-Stores} & = & \text{Variables} \rightharpoonup \text{Bt-Values} \\ \Gamma & \in & \text{Block-Maps} & = & \text{Block-Labels} \rightharpoonup \text{Basic-Blocks} \end{array}$$

Figure 6.6: Maximally polyvariant binding-time analysis: relation $\rightarrow_{bta}$

6.5.1 Maximally Polyvariant Binding-Time Analysis

The BTA does not use the static values in store σ. The BTA relies only on the static and dynamic tags in bt-store β to determine the set of reachable bt-configurations. A bt-configuration is a pair (l,β) consisting of a label l and a bt-store β.

Figure 6.6 defines a transition relation $\rightarrow_{bta}$ between stacks of bt-configurations. A judgement $\vdash_\Gamma r \rightarrow_{bta} r'$ represents a transition from a stack r to a stack r' in a program Γ. The rules extract the computation of the static and dynamic tags from the online specialization phase (Figure 6.4). In fact, the only rules that determine a new bt-store are the rules for assignments and returns. The jump rule has to return all labels because there are no static values available to determine the outcome of the test.

We can define the set of reachable bt-configurations as follows:

Definition 7.

$$Poly_{bta} = \{\, (l,\beta) \mid (l_0,\beta_0):\varepsilon \rightarrow^*_{bta} (l,\beta):r \,\}$$

Set $Poly_{bta}$ is always finite because a program has finitely many program points, finitely many variables, and there are finitely many bt-values. We shall separate the set of reachable bt-configurations from the syntactic annotation of a source program. (Traditionally, programs are annotated with the bt-information obtained by the BTA for presenting it to the specialization phase, as well as to the user. We use the relation as a ‘database’ to which the specialization phase refers.)

By comparing the rules with those in Figure 6.4, it is not hard to see that relation $\rightarrow_{bta}$ calculates the same bt-store β' for the block labelled l and bt-store β as the online transition $\rightarrow_{on}$ would for the same block and the same bt-store (this relation will be explored further in Section 6.7). The BTA is *polyvariant* because each block can be associated with multiple bt-stores. There is no merging or generalization of bt-stores.[4] To express this fact, we call the bt-relation *maximally polyvariant* [CGL00] (short *polymax*). A distinguishing feature is that *all* program points are handled in a polyvariant manner during the analysis, including assignment blocks, procedure calls and returns. Existing monovariant and polyvariant BTAs compromise on some or all of these points. The tradeoff for this precision is that the number of different bt-stores the maximally polyvariant analysis must handle may grow for each program point (in the worst case, exponentially: a bt-store with n variables possibly has 2^n different S/D patterns). Thus, the accuracy of this BTA will clearly come at the expense of its efficiency.

A bt-configuration can be viewed as an abstraction of a configuration in partial evaluation [Rom88] (it represents a set of pe-configurations). The following relation holds for a maximally polyvariant BTA (see also Section 6.7).

Property 1.

$$\{\,(l,\beta)\mid(l,(\beta,\sigma))\in Poly_{on}\,\}\subseteq Poly_{bta}$$

It is vital for an offline partial evaluator that the BTA used is *congruent* [Jon88]. In essence, this is a correctness criterion for BTAs. Our maximally polyvariant BTA is congruent because any variable that depends on a dynamic value is itself classified as dynamic.

Example 3. *Consider again the program and initial pe-store of Example 2. If we apply the maximally polyvariant BTA to these, we get*

$$\begin{aligned}Poly_{bta}=\{\ &(\texttt{loop},[\texttt{i}\mapsto S,\texttt{a}\mapsto S,\texttt{z}\mapsto D]),(\texttt{loop},[\texttt{i}\mapsto S,\texttt{a}\mapsto S,\texttt{z}\mapsto S]),\\ &(\texttt{test},[\texttt{i}\mapsto S,\texttt{a}\mapsto S,\texttt{z}\mapsto D]),(\texttt{test},[\texttt{i}\mapsto S,\texttt{a}\mapsto S,\texttt{z}\mapsto S]),\\ &(\texttt{setz},[\texttt{i}\mapsto S,\texttt{a}\mapsto S,\texttt{z}\mapsto D]),(\texttt{setz},[\texttt{i}\mapsto S,\texttt{a}\mapsto S,\texttt{z}\mapsto S]),\\ &(\texttt{done},[\texttt{i}\mapsto S,\texttt{a}\mapsto S,\texttt{z}\mapsto D]),(\texttt{done},[\texttt{i}\mapsto S,\texttt{a}\mapsto S,\texttt{z}\mapsto S]),\\ &(\texttt{retn},[\texttt{i}\mapsto S,\texttt{a}\mapsto S,\texttt{z}\mapsto D]),(\texttt{retn},[\texttt{i}\mapsto S,\texttt{a}\mapsto S,\texttt{z}\mapsto S])\ \}\end{aligned}$$

[4] In contrast, most partial evaluators control the degree of polyvariance by merging several bt-stores into one. This means that their rules for computing bt-configurations may be more conservative than their online counterpart.

Using this information, it is not hard to see that, for this program and pe-store, we have $Poly_{on} = Poly_{off}$.

Example 4. *A uniform BTA would associate at most one bt-store with each basic block. In the cases where our BTA finds more than one, the uniform analysis unifies bt-stores by generalization, i.e., turning appropriate variables dynamic.*

In the situation of Example 2 a uniform BTA would likely return this set of configurations:

$$\begin{aligned} Poly_{unibta} = \{ & (\texttt{loop}, [\texttt{i} \mapsto S, \texttt{a} \mapsto S, \texttt{z} \mapsto D]), \\ & (\texttt{test}, [\texttt{i} \mapsto S, \texttt{a} \mapsto S, \texttt{z} \mapsto D]), \\ & (\texttt{setz}, [\texttt{i} \mapsto S, \texttt{a} \mapsto S, \texttt{z} \mapsto D]), \\ & (\texttt{done}, [\texttt{i} \mapsto S, \texttt{a} \mapsto S, \texttt{z} \mapsto D]), \\ & (\texttt{retn}, [\texttt{i} \mapsto S, \texttt{a} \mapsto S, \texttt{z} \mapsto D]) \} \end{aligned}$$

If we used such an analysis in our offline partial evaluator, the assignment `z := i` *(when* $\texttt{a} = \texttt{1}$ *and* $\texttt{i} = \texttt{1}$*) could not be performed at specialization time, and we would have that* $Poly_{on} \neq Poly_{off}$.

6.6 Block Specialization

We now move to the process of constructing the residual program. The reader should note that this process is independent of whether the preceding collection of reachable configurations was done by an online or offline strategy.

As mentioned in Section 6.3.2, the actual code of the residual program will consist of specialized basic blocks of the source program. The main idea is to produce a specialized basic block for each reachable state $(l,(\beta,\sigma))$. This is done by specializing block $(l\mathbf{:}\ a\ j)$ *wrt* the static values available in the pe-store (β,σ).

Since we have recursive procedures, the generation of residual labels is a bit more involved. Let π range over pe-stores, i.e., $\pi = (\beta,\sigma)$, and define $proc(r)$ to be the pe-store in the top configuration of stack r, i.e., $proc((l',\pi'):r') = \pi'$. As a special case, if the stack is empty, $proc(\varepsilon)$ is a unique “dummy” pe-store.

6.6.1 Labels in the Residual Program

Our system produces a unique residual block for each (l,π) and π' if the collection phase encounters a stack $(l,\pi):r$ where $proc(r) = \pi'$. We label the residual block for (l,π) and π' by a triple of the form $\pi'{\cdot}l{\cdot}\pi$. Part of that triple is the pe-store π' with which the procedure was called to which the block labeled l belongs and the pe-store π *wrt* which that block is specialized. Together this achieves procedure cloning: for each unique pe-store π', a unique residual procedure is generated.

Assignment

$$\frac{\forall i\,.\,\beta(x_i)=S \quad \forall i\,.\,\sigma(x_i)=v_i \quad [\![o]\!](v_1\ldots v_n)=v}{(l,(\beta,\sigma)):r \vdash_{cg} x := o(x_1\ldots x_n) \Rightarrow x := o_v()}$$

$$\frac{\exists i\,.\,\beta(x_i)=D}{(l,(\beta,\sigma)):r \vdash_{cg} x := o(x_1\ldots x_n) \Rightarrow x := o(x_1\ldots x_n)}$$

Jump

$$\frac{\forall i\,.\,\beta(x_i)=S \quad (l,(\beta,\sigma)):r \rightarrow_{pe} (l_k,\pi_k):r}{(l,(\beta,\sigma)):r \vdash_{cg} \textbf{case}\ t(x_1\ldots x_n)\ l_1\ldots l_m \Rightarrow \textbf{case}\ t_{\mathsf{true}}()\ proc(r)\cdot l_k\cdot\pi_k}$$

$$\frac{\exists i\,.\,\beta(x_i)=D \quad \forall k\,.\,(l,(\beta,\sigma)):r \rightarrow_{pe} (l_k,\pi_k):r}{(l,(\beta,\sigma)):r \vdash_{cg} \textbf{case}\ t(x_1\ldots x_n)\ l_1\ldots l_m \Rightarrow \textbf{case}\ t(x_1\ldots x_n)\ proc(r)\cdot l_1\cdot\pi_1\ldots proc(r)\cdot l_m\cdot\pi_m}$$

Basic block

$$\frac{\Gamma(l)=a\ j \quad (l,\pi):r \vdash_{cg} a \Rightarrow a' \quad (l,\pi):r \vdash_{cg} j \Rightarrow j'}{\vdash_{\Gamma} (l,\pi):r \rightarrow_{cg} proc(r)\cdot l\cdot\pi\textbf{:}\ a'\ j'}$$

$$\frac{\Gamma(l)=\textbf{call}\ l_1\ l_2}{\vdash_{\Gamma} (l,\pi):r \rightarrow_{cg} proc(r)\cdot l\cdot\pi\textbf{:}\ \textbf{call}\ \pi\cdot l_1\cdot\pi\ proc(r)\cdot l_2}$$

$$\frac{\Gamma(l)=\textbf{return}\,x\ l'' \quad (l,\pi):r \rightarrow_{pe} (l',\pi'):r'}{\vdash_{\Gamma} (l,\pi):r \rightarrow_{cg} proc(r)\cdot l\cdot\pi\textbf{:}\ \textbf{return}\,x\ l''\cdot\pi'}$$

Semantic Values

l	$\in$	Labels	$=$	Block-Labels $\cup$ $\{\mathsf{nil}\}$
σ	$\in$	Stores	$=$	Variables $\rightharpoonup$ Values
β	$\in$	Bt-Stores	$=$	Variables $\rightharpoonup$ Bt-Values
Γ	$\in$	Block-Maps	$=$	Block-Labels $\rightharpoonup$ Basic-Blocks
π	$\in$	Pe-Stores	$=$	Bt-Stores $\times$ Stores

Figure 6.7: Block specialization (the relation $\rightarrow_{pe}$ is either $\rightarrow_{on}$ or $\rightarrow_{off}$)

This addition is technically necessary, but it also prevents, in some cases, the sharing of blocks between different procedures. The two pe-stores, π and π', are contained in the top two configurations of the stack. The reader should still think of the dot-operator as a concatenation of labels, only in the residual program, pe-stores are (parts of) labels, too.

In the residual version of a procedure call, the return-label (i.e., the label to proceed to after the procedure returns) will be a concatenation $\pi'\cdot l$ of the first two parts of our label triples. Although this is not a label of a block in the residual program, it will not go wrong. This is because all procedure returns will use an offset of the form $l'{\cdot}\pi$ and when we define

$$(\pi'{\cdot}l){\cdot}(l'{\cdot}\pi) = \pi'{\cdot}(l{\cdot}l'){\cdot}\pi$$

the right-hand side of this equation *will* be the label of a block in the residual program. A return-statement may give raise to several return-statements in a residual program if that statement is reachable by different pe-stores. Each return-statement has a different offset depending on the pe-store by which it is reached. Thus, each of them will return to a different block. We call this *polyvariant specialization of procedure exits*. In traditional partial evaluators, specialization of procedure entries is polyvariant, but the specialization of procedure exists is monovariant. This can result in a loss of important static information when we return from a procedure. Our partial evaluators do not suffer from this information loss. An example will be shown in Section 6.8.

6.6.2 Inference Rules for Code Generation

Figure 6.7 presents all rules for code generation. A judgement $\vdash_\Gamma r \rightarrow_{cg} b$ represents the generation of a basic block b given a configuration stack r and a program Γ. The relation $\rightarrow_{pe}$ is either $\rightarrow_{on}$ or $\rightarrow_{off}$, depending on which kind of specializer we are generating code for.

To build a constant from a value v, we write the nullary operator $o_v()$, where $[\![o_v]\!]() = v$. This operator is used to lift static values during code generation. It is used when the operator $o(x_1 ... x_n)$ on the right-hand side of an assignment can be reduced to a value v. Similarly, a test that always selects the first label of a label sequence is denoted by $t_{\mathsf{true}}()$ and we have $[\![t_{\mathsf{true}}]\!](l) = l$.

When a test $t(x_1 ... x_n)$ in a jump is static, then the test can be decided and we generate an unconditional jump inserting $t_{\mathsf{true}}()$ as a test; otherwise, when the test is dynamic, we generate a jump containing m new labels ($1 \leq k \leq m$). Thus, the residual jump generated when the test is dynamic has the same number of labels as the original jump. When the test is static, the residual jump contains exactly one label. It is equivalent to a goto-statement. Our relations $\rightarrow_{on}$ or $\rightarrow_{off}$ satisfy these two properties (we also have $\pi_1 = ... = \pi_m$ in the fourth rule of Figure 6.7).

The total residual program is the collection of basic blocks b such that $\vdash_\Gamma r \rightarrow_{cg} b$ where $(l_0, (\beta_0, \sigma_0)) : \varepsilon \rightarrow^*_{pe} r$.

6.6.3 The Residual Program

In what sense is a source program optimized after partial evaluation? As can be seen from the rules, operators and tests are replaced by constants if they depend

Residual program:

```
0_loop_0: a := 2
          GOTO 0_test_1
0_test_1: i := 1
          GOTO 0_loop_2
0_loop_2: a := 1
          GOTO 0_test_3
0_test_3: i := 0
          GOTO 0_setz_4
0_setz_4: z := 0
          GOTO 0_loop_5
0_loop_5: a := 1
          GOTO 0_done_5
0_done_5: z := 1
          GOTO 0_retn_6
0_retn_6: RETURN z end_7
```

After postprocessing:

```
0_test_3: i := 0
          GOTO 0_loop_5
0_loop_5: a := 1
          GOTO 0_done_5
0_done_5: z := 1
          GOTO 0_retn_6
0_retn_6: RETURN z end_7
```

Figure 6.8: Residual program for *monitor-loop* (`i`=2, `a`=4, `z`=*D*)

only on static values; multi-way jumps are reduced to one-way jumps (*i.e.*, unconditional jumps). Partial evaluation prunes unnecessary branches and specializes each reachable block *wrt* each static store that is flowing into that block.

In practice, we also apply a technique of transition compression to eliminate one-way transitions and inline constants. Dead variable elimination removes redundant assignments. However, due to the simple syntax of our flowchart language, these optimizations do not always make sense (arguments of operators must be variables; assignments cannot be collected in a single basic block) and to perform them effectively one would have to translate the programs to a less restricted language.

Example 5. *Figure 6.8 shows the residual program generated by the rules in Figure 6.7 for program* monitor-loop *(Figure 6.2). Postprocessing can further improve the residual program by eliminating assignments to dead variables and compressing redundant jumps.*

6.7 Equivalence of the Two Partial Evaluators

In this section, we shall prove the main technical result of this paper: That the online and offline partial evaluators defined above are *functionally equivalent*.

Hence, one cannot be said to be more accurate than the other. More precisely, we shall prove the following:

Theorem 1. *For any program* Γ *and any call stacks* r,r' *the following holds:*

$$\begin{gathered}\vdash_\Gamma r \rightarrow_{on} r' \\ \Longleftrightarrow \\ \vdash_\Gamma r \rightarrow_{off} r'\end{gathered}$$

The relations $\rightarrow_{on}$, $\rightarrow_{off}$, and $\rightarrow_{bta}$ are defined in Figures 6.4, 6.5, and 6.6, respectively. Note that the result builds on the fact that we are using a maximally polyvariant BTA ($\rightarrow_{bta}$) in the offline system ($\rightarrow_{off}$). As Example 4 illustrated, the theorem would *not* be valid if we used a uniform BTA.

The theorem clearly holds when r is empty. If r is non-empty, let $r = (l,(\beta,\sigma)):r''$. We shall prove the theorem by case analysis on the basic block labelled l. We examine each type of basic block that can be defined by $\Gamma(l)$, viz., assignment block, call block and return block. Each will be treated in its own subsection below.

6.7.1 Case: Assignment Block

In order to prove our theorem in this case, we observe that the offline transitions for assignment blocks are defined by the conjunction of three conditions. The following fact is just a rewritten version of the first basic block rule for offline specialization:

Observation 1. *For any program* Γ*, any labels* l,l'*, any bt-stores* β,β' *and any stores* σ,σ'*, let* a *and* j *be such that* $\Gamma(l) = a\ j$*. Then the following holds for all* r*:*

$$\begin{gathered}\vdash_\Gamma (l,(\beta,\sigma)):r \rightarrow_{off} (l',(\beta',\sigma')):r \\ \Longleftrightarrow \\ (\beta,\sigma) \vdash_{off} a \Rightarrow \sigma' \quad \wedge \\ (\beta,\sigma) \vdash_{off} j \Rightarrow l' \quad \wedge \\ \vdash_\Gamma (l,\beta):\bar{r} \rightarrow_{bta} (l',\beta'):\bar{r}\end{gathered}$$

The next three lemmata show that online transition implies each of these three conditions, thus online transition implies offline transition.

Lemma 1. *For any program* Γ*, any labels* l,l'*, any bt-stores* β,β' *and any stores* σ,σ'*, let* a *and* j *be such that* $\Gamma(l) = a\ j$*. Then the following holds for all* r*:*

$$\begin{gathered}\vdash_\Gamma (l,(\beta,\sigma)):r \rightarrow_{on} (l',(\beta',\sigma')):r \\ \Longrightarrow \\ (\beta,\sigma) \vdash_{off} a \Rightarrow \sigma'\end{gathered}$$

Proof. Assume $\vdash_\Gamma (l,(\beta,\sigma)):r \rightarrow_{on} (l',(\beta',\sigma')):r$. By the first basic block rule for online specialization, we must also have $(\beta,\sigma) \vdash_{on} a \Rightarrow (\beta',\sigma')$. Let a be given by $x := o(x_1 ... x_n)$. By the two assignment rules for $\vdash_{on}$ we have one of two cases:

$\boxed{\textbf{Case } \forall i\,.\,\beta(x_i) = S}$ We conclude that $\sigma' = \sigma[x \mapsto [\![o]\!](\sigma(x_1) ... \sigma(x_n))]$ (which is well-defined). Moreover, we see that the first assignment rule for $\vdash_{off} \Rightarrow$ applies and yields $(\beta,\sigma) \vdash_{off} a \Rightarrow \sigma'$ with the same σ'.

$\boxed{\textbf{Case } \exists i\,.\,\beta(x_i) = D}$ We conclude that $\sigma' = \sigma|_x$. Moreover, we see that the second assignment rule for $\vdash_{off}$ applies and yields $(\beta,\sigma) \vdash_{off} a \Rightarrow \sigma'$ with the same σ'. □

Lemma 2. *For any jump j, any bt-store β, any store σ and any label l, the following holds:*

$$(\beta,\sigma) \vdash_{on} j \Rightarrow l \\ \Longleftrightarrow \\ (\beta,\sigma) \vdash_{off} j \Rightarrow l$$

Proof. Trivial, as the jump rules for online and offline specialization are exactly the same. □

Lemma 3. *For any program Γ, any labels l, l' such that $\Gamma(l) = a\ j$, any bt-stores β, β' and any stores σ, σ', the following holds for all r:*

$$\vdash_\Gamma (l,(\beta,\sigma)):r \rightarrow_{on} (l',(\beta',\sigma')):r \\ \Longrightarrow \\ \vdash_\Gamma (l,\beta):\overline{r} \rightarrow_{bta} (l',\beta'):\overline{r}$$

Proof. Assume $\vdash_\Gamma (l,(\beta,\sigma)):r \rightarrow_{on} (l',(\beta',\sigma')):r$. By the block rule for the BTA, we need to prove that

$$\beta \vdash_{bta} a \Rightarrow \beta' \tag{6.1}$$

and

$$\beta \vdash_{bta} j \Rightarrow l' \tag{6.2}$$

We prove Statement (6.1) by cases. Let a be given by $x := o(x_1 ... x_n)$. We must have one of the following two cases:

$\boxed{\textbf{Case } \forall i\,.\,\beta(x_i) = S}$ We conclude that $\beta' = \beta[x \mapsto S]$. Moreover, we see that the first assignment rule for $\vdash_{bta}$ applies and yields $\beta \vdash_{bta} a \Rightarrow \beta'$ with the same β'.

$\boxed{\textbf{Case } \exists i\,.\,\beta(x_i) = D}$ We conclude that $\beta' = \beta[x \mapsto D]$. Moreover, we see that the second assignment rule for $\vdash_{bta}$ applies and yields $\beta \vdash_{bta} a \Rightarrow \beta'$ with the same β'. Thus, Statement (6.1) holds.

Now, let j be given by **case** $t(x_1 \ldots x_n)\ l_1 \ldots l_m$ and observe that in both jump rules for online specialization, label l' is among $l_1, \ldots, l_m$. By the jump rule for $\vdash_{bta}$, Statement (6.2) holds. $\square$

As mentioned above, Lemmata 1 to 3 have as consequence the following:

Theorem 2. *For any program Γ, any labels l, l' such that $\Gamma(l) = a\ j$, any bt-stores β, β' and any stores σ, σ', the following holds for all r:*

$$\begin{array}{c} \vdash_\Gamma\ (l,(\beta,\sigma)) : r \rightarrow_{on} (l',(\beta',\sigma')) : r \\ \Longrightarrow \\ \vdash_\Gamma\ (l,(\beta,\sigma)) : r \rightarrow_{off} (l',(\beta',\sigma')) : r \end{array}$$

We now turn to the reverse implication – that offline transition implies online transition. Again, we take our outset in the three conditions.

Theorem 3. *For any program Γ, any labels l, l' such that $\Gamma(l) = a\ j$, any bt-stores β, β' and any stores σ, σ', the following holds for all r:*

$$\begin{array}{c} (\beta,\sigma) \vdash_{off} a \Rightarrow \sigma' \quad \wedge \\ (\beta,\sigma) \vdash_{off} j \Rightarrow l' \quad \wedge \\ \vdash_\Gamma\ (l,\beta) : \bar{r} \rightarrow_{bta} (l',\beta') : \bar{r} \\ \Longrightarrow \\ \vdash_\Gamma\ (l,(\beta,\sigma)) : r \rightarrow_{on} (l',(\beta',\sigma')) : r \end{array}$$

Proof. Assume that the left side of the implication holds. By the block rule for online specialization, we need to prove:

$$(\beta,\sigma) \vdash_{on} a \Rightarrow (\beta',\sigma') \tag{6.3}$$

and

$$(\beta,\sigma) \vdash_{on} j \Rightarrow l' \tag{6.4}$$

Statement (6.4) follows directly from Lemma 2.

We prove Statement (6.3) by cases. Let a be given by x **:=** $o(x_1 \ldots x_n)$. We must have one of the following two cases:

$\boxed{\textbf{Case}\ \forall i\,.\,\beta(x_i) = S}$ We conclude that $\sigma' = \sigma[x \mapsto [\![o]\!](\sigma(x_1) \ldots \sigma(x_n))]$ (which is well-defined). Furthermore, $(l,\beta) : \bar{r} \rightarrow_{bta} (l',\beta') : \bar{r}$ must hold using the first assignment rule for $\vdash_{bta}$, hence $\beta' = \beta[x \mapsto S]$. By these observations, the first assignment rule for online specialization applies and yields Statement (6.3) with the same β' and σ'.

$\boxed{\textbf{Case}\ \exists i\,.\,\beta(x_i) = D}$ We conclude that $\sigma' = \sigma|_x$. Furthermore, $(l,\beta) : \bar{r} \rightarrow_{bta}$ $(l',\beta') : \bar{r}$ must hold using the second assignment rule for $\vdash_{bta}$, hence $\beta' = \beta[x \mapsto D]$.

By these observations, the second assignment rule for online specialization applies and yields Statement (6.3) with the same β' and σ'. $\square$

From Observation 1 and Theorems 2 and 3 we finally derive our main statement:

Corollary 1. *For any program* Γ*, any labels* l, l' *such that* $\Gamma(l) = a\ j$*, any bt-stores* β, β' *and any stores* σ, σ'*, the following holds for all* r*:*

$$\begin{gathered}\vdash_\Gamma (l,(\beta,\sigma)):r \rightarrow_{on} (l',(\beta',\sigma')):r\\ \Longleftrightarrow\\ \vdash_\Gamma (l,(\beta,\sigma)):r \rightarrow_{off} (l',(\beta',\sigma')):r\end{gathered}$$

6.7.2 Case: Call Block

This case is trivial as the rules for calls are identical in the online and offline semantics.

6.7.3 Case: Return Block

To show the Theorem in this case, observe the following:

Observation 2. *Let* Γ*, l, x and* l'' *be such that* $\Gamma(l) = \textbf{return}\ x\ l''$*. For all* β*,* l'*,* β'*,* β'' *and r:*

$$\begin{gathered}\vdash_\Gamma (l,\beta):(l',\beta'):\bar{r} \rightarrow_{bta} (l'{\cdot}l'',\beta''):\bar{r}\\ \Longleftrightarrow\\ \beta'' = \beta'[x \mapsto \beta(x)]\end{gathered}$$

This can be seen immediately from the BTA rule for procedure returns.

Given this observation, the rules for procedure returns in the online and offline semantics must be equivalent, which proves our Theorem in this case.

6.8 Example: Achieving Constant Folding While Specializing an Interpreter

We will now show how the optimization of constant folding can be achieved by specializing an expression interpreter – even with an offline partial evaluator. In [Bul93] this was obtained by cleverly rewriting the given expression interpreter to incorporate explicit tests on SD-values, dispatching the flow of control into static/dynamic versions of the procedures in the source program. Together with a monovariant partial evaluator, this manually produced *polyvariant expansion* [TS96] of the source program and led to the same optimization that is achievable by an online specializer. Given the results in Section 6.7, namely that our

```
PROGRAM (e env) (evalExp)

; PROCEDURE evalExp :: Exp x Env -> Val

   evalExp:  CASE (isCst? e) getCst isVarOp    ; constant
   isVarOp:  CASE (isVar? e) getVar getOp      ; variable, operator
   getCst:   v := (fetchCst e)                 ; get constant
             GOTO exitExp
   getVar:   v := (lookupVar e env)            ; lookup variable
             GOTO exitExp
   getOp:    op := (fetchOp e)                 ; get operator
             GOTO getArgs
   getArgs:  es := (fetchArgs e)               ; get arguments
             GOTO doArgs
   doArgs:   CALL evalExps doApp               ; eval arguments
   doApp:    v := (applyOp op vs)              ; apply operator
             GOTO exitExp
   exitExp:  RETURN v nil                      ; return value

; PROCEDURE evalExps :: [Exp] x Env -> [Val]

   evalExps: CASE (null? es) initVs getE       ; expressions
   initVs:   vs := '()                         ; empty vs
             GOTO exitExps
   getE:     e := (car es)                     ; get first exp
             GOTO getEs
   getEs:    es := (cdr es)                    ; get rest exps
             GOTO doE
   doE:      CALL evalExp doEs                 ; eval first exp
   doEs:     CALL evalExps consVs              ; eval rest exps
   consVs:   vs := (cons v vs)                 ; cons values
             GOTO exitExps
   exitExps: RETURN vs nil                     ; return values
```

Figure 6.9: Expression interpreter *evalExp*

```
PROGRAM (env) (0_evalExp_0)   ; computes (+ x 3) * 5

   0_evalExp_0: v := (lookupVar 'x env)
                GOTO 1b_doEs_1d
   1b_doEs_1d:  vs := (cons v '(3))
                GOTO 5_doApp_1b
   5_doApp_1b:  v := (applyOp '+ vs)
                GOTO 3_doEs_5
   3_doEs_5:    vs := (cons v '(5))
                GOTO 0_doApp_3
   0_doApp_3:   v := (applyOp '* vs)
                GOTO 0_exitExp_3
   0_exitExp_3: RETURN v _0
```

Figure 6.10: Residual program for *evalExp* (`e=((x+3)*(7-2))`, `env`=*D*) after transition compression and inlining.

offline partial evaluator with a maximally polyvariant BTA is functionally equivalent to our online partial evaluator, we know that this transformation should be achievable by an offline partial evaluator without modifying the source program.

The expression interpreter in Figure 6.9 consists of two recursive procedures, which we call `evalExp` and `evalExps` according to their first block. Given an environment `env`, they evaluate an expression and a list of expressions, respectively. The header of the program tells us that the interpreter has two inputs (`e`, `env`) and that the initial label is `evalExp`.[5] The program is identical to the one in [Bul93] except for the use of an imperative instead of a functional language.

Consider specializing the interpreter with respect to a static expression `e=((x + 3) * (7 - 2))` and a dynamic environment `env`=*D*. A monovariant BTA requires that any single block in the interpreter have only one specialization behavior. This prohibits the offline specializer from taking advantage of the structure of the expression being interpreted, yielding an non-optimizing specialization of the expression. In particular, in `e` subtraction `(7 - 2)` cannot be performed. Consider block `doApp`. Since the argument values `vs` of the arithmetic operators are not always known, `applyOp` cannot be applied and is annotated as dynamic. Thus, assigned variable `v` becomes dynamic, and the value returned in block `exitExp` is dynamic, too. Thus, any call to `evalExp` will return a dynamic value at specialization time. This means that a constant folding optimization cannot be performed at specialization time.

[5]The procedure headers are only textual comments; syntactically required dummy assignments are omitted from blocks `evalExp`, `isVarOp`, and `evalExps`.

This is in contrast to the results obtained by our partial evaluators. We have an implementation that includes all parts of the partial evaluators we have defined in the earlier sections. The partial evaluators were applied to the expression interpreter, and the results were, of course, identical. The common result can be seen in Figure 6.10. This is the best specialization result one could hope for. The problems discussed above were not relevant to the offline system because the polyvariant BTA in essence incorporates polyvariant expansion as needed.

6.9 Extensions and Limitations

Intentionally changing a variable's classification from static to dynamic is called *generalization*. There are different forms of generalization, and we shall discuss two types of generalization and argue why the true power of online partial evaluation comes from *online generalization*.

6.9.1 Offline Generalization

We first discuss what we call *offline generalization*. To tell the specializer to generalize a variable x (turn it dynamic), we can add an operator **gen** to the source language. In F, we add this operator to the syntax of assignments in Figure 6.1:

$$a \quad ::= \quad \dots \mid x := \mathbf{gen}\, x$$

Such an operator may be used by static analyses, *e.g.*, Glenstrup's and Jones's analysis, for ensuring termination of offline partial evaluation [GJ96]. It is also common to have such an operator when using online partial evaluation, so that the user may manually instruct the specializer to avoid possible finite or infinite code explosion [Hat99, JG02].

The operator is just the identity function in the semantics of F.

Figure 6.11 shows the rules for the online and offline partial evaluator. They change the variable value in the pe-store from static to dynamic. It is not hard to see that the operator **gen** does not change the assumptions in the proof and so our result also holds when the operator is included in the language. Both partial evaluators are controlled in the same way by that operator and their residual programs are identical.

6.9.2 Online Generalization

We speak of *online generalization* when the values computed during partial evaluation can effect the choice of actions taken for dynamizing values. Consider the following program fragment:

Online collection phase

$$\frac{}{(\beta,\sigma) \vdash_{on} x := \mathbf{gen}\, x \Rightarrow (\beta[x \mapsto D], \sigma|_x)}$$

Offline collection phase

$$\frac{}{(\beta,\sigma) \vdash_{off} x := \mathbf{gen}\, x \Rightarrow \sigma|_x}$$

Binding-time analysis

$$\frac{}{\beta \vdash_{bta} x := \mathbf{gen}\, x \Rightarrow \beta[x \mapsto D]}$$

Figure 6.11: Generalization operator in online and offline partial evaluation

```
loop: CASE (i = 0) done next
next: ...
      CASE (br) done dec
dec : i := i - 1
      GOTO loop
done: ...
```

Suppose specialization will reach the block marked `loop` with variable `i` having a static value of $2^{32} - 1$ while variable `br` is dynamic (the programmer must count on `br` to become true sooner or later).

The maximally polyvariant specializers (online and offline) constructed in the earlier sections of this paper will both suffer from an unreasonable code explosion when generating the specialization of this program, unfolding the above with `i` taking on all (unsigned) 32-bit integer values. Offline termination analyses will normally not prevent this as the process should, after all, terminate. Estimating the size of values by offline analysis can be difficult or even impossible.

By contrast, an online specializer may incorporate a simple scheme to avoid unreasonable code explosion: generalize variable `i` when the number of copies of the loop reaches some predetermined upper bound. Such a decision based on the specialization history is "very online" in its nature and cannot be directly simulated by an offline partial evaluator in the sense of Section 6.7. An even more drastic example is where online partial evaluators use criteria on the specialization history to guide generalizations that ensure termination. This history is not available to the BTA of an offline system and, again, we have "very online" decisions.

To conclude, while an offline specializer can be as accurate as an online specializer in propagating static values, the latter may be able to choose generalizations on the basis of more information (*e.g.*, static values, specialization history). Our point is that, while this is very important in practice (a clear advantage for an online partial evaluator), it cannot help in finding more static information. Hence, an online specializer is stronger only in the sense that it is able to make more well-informed decisions regarding which program parts it should *not* unfold. In other words, the strength of online partial evaluators is the intentional loss of information.

6.10 Related Work

Since the conception of offline partial evaluation [JSS85], much research has been devoted to improving the results of offline partial evaluation. In this paper, we disregarded practical issues to get to the core of the main question, namely, what are the theoretical limits of offline partial evaluation? For this, we gave two reference systems based on constant propagation and *polyvariant specialization* [Bul84] (calls are folded if they have identical pe-stores).

Our online system extends the online partial evaluator for flowcharts [Hat99] with recursive procedures; otherwise, the two partial evaluators are identical modulo a syntactical simplification of our source language. Both correspond to the online partial evaluators [RW93, CK95], and are less conservative than the online partial evaluator [Mey99] which inserts explicators at suspended branches of conditionals. Our offline system extends the offline partial evaluators for flowcharts [Jon88, GJ91, Hat99] with recursive procedures and a *maximally polyvariant* BTA [CGL00].

An early example of a powerful, polyvariant BTA is given in [Con93]. Though it is not maximally polyvariant as our BTA (bt-stores are merged at procedure exits and after conditionals), this analysis obtains parametrized polyvariance and treats both higher-order constructs and partially static structures.

How can online effects be achieved by offline systems? Three techniques are known: (i) modifying the partial evaluator, (ii) modifying the source programs, and (iii) inserting an interpreter between a partial evaluator and a source program.

In the first category, we find works which refine the specialization phase and/or the BTA. One technique is continuation-based specialization [Bon92, LD94]; another is increasing the accuracy of constant propagation by a pointwise or polyvariant BTA [RG92, AC94, HN99, Asa99]. Offline systems can be combined with online features in mixline systems [JGS93] (see [SK00]).

In the second category, we find binding-time improvements: semantics-preserving transformations that are applied to a source program before specialization. Numerous binding-time improvements are described in the literature [CD91a, Bon93, JGS93]. It was shown [Glü02] that, for every Jones-optimal offline partial evaluator with static expression reduction, there exists a binding-time improver that allows it to achieve the same residual-program efficiency of any online partial evaluator.

In the last category, there are works on bootstrapping higher-order partial evaluators from first-order ones [SGT96], and simulating deforestation and unification-based systems by offline partial evaluators [GJ94].

6.11 Conclusion

We have introduced two specialization systems, representing online and offline partial evaluation, respectively. The two systems treat a flowchart language with recursive procedures. We proved the two to be *functionally equivalent* and gave an example based on the results of our implementation of the systems. Our result is contrary to common belief: that offline partial evaluation is weaker in the sense that it cannot be as accurate – detect as much static information – as online partial evaluation.

We compared offline and online generalization, and found that the real strength of online systems as compared to offline systems is that the former can have more fine-grained strategies regarding when to intentionally *forget* static values by online generalization. We hope that these results clarify the consequences of choosing either “line” when designing a practical partial evaluation system in the future. Knowing what can be achieved in principle makes it easier to choose a technique on practical and pragmatic grounds (*e.g.*, efficiency, usability).

In [Ruf93, page 19], a term is coined, applying to any online specializer, the effect of which might be achieved by offline methods. Such online specializers are called *vacuously online* (“online specializers that really aren’t”). On the basis of Section 6.7, we see that this predicate does not concern accuracy – the static reductions a specializer is able to perform – but rather the *informed choice of imprecision* as discussed in Section 6.9.

6.11.1 The Designer’s Choice

What factors should be considered, when the designer of a partial evaluator decides whether to make his system online or offline? We suggest the following:

Ease of implementation As mentioned in Section 6.9, some features may be easier to realize using online rather than offline specialization (or perhaps vice versa).

Ease of control Often the most useful systems are semi-automatic, allowing the user to influence on which transformations the system performs. The separate BTA phase in offline systems can be used to provide useful feedback to the user ([GMS99]).

Sophisticated automatic choices of specialization precision Depending on the intended use of a partial evaluator, the observations in Section 6.9.2 may force the designer to choose an online approach.

In relation to the latter point: The demonstration that BTAs that can form the basis of surprisingly accurate offline partial evaluators exist serves to illustrate that the design space of BTAs is indeed large. This supports the earlier thesis [Con93, CGL00] that within the framework of offline partial evaluation, fixing a BTA strategy at design-time can seriously limit the usefulness of a system.

6.11.2 Future Work

While the language we treat is quite expressive, we would like to investigate the limits of polyvariant BTA in the presence of non-atomic data types (such as arrays). Early investigations seem to indicate that there are important cases when an offline approach may simulate the accuracy of online systems even though these generally work much better on partially static data structures. One central aspect of such an investigation is the definition of what constitutes an automatic binding-time improvement, in particular: How much must the result of the improvement resemble the source program? In a simple polyvariant expansion, all lines of the improved program are essentially copies of lines of the source, but if we do not make this relation a requirement of binding-time improvements, some very powerful systems may be constructed along these lines.

Acknowledgements

We would like to thank Kenichi Asai, Neil D. Jones, Torben Æ. Mogensen and Sergei Romanenko for many valuable comments on an earlier version of this paper. We also thank the anonymous referees for their useful feedback.

Part III

MORI: an application of domain-specific languages

7 - The becoming of MORI/SQL

Keeping a brave face
in circumstances is impossible
Cannot describe so many decisions
It's impossible
"2:1", Elastica

This chapter and the two that follow regard an industrial project in which a domain-specific language was invented by the author. The language, called *MORI/SQL*, facilitates smooth customization of a (specific) complicated software system with the IT systems at any given customer site.

The specific software system that we[1] consider is in the product line of *Visanti A/S*. Visanti is a Danish IT company the goal of which is to solve problems related to decision support, knowledge management and document handling in medium- to large-size companies. Our language and its compiler form an customization tool for one of Visanti's products.

Put very simply, the product consists of a log server, a number of logging clients and a number of clients for querying the server. This system collects and organizes information related to human interaction with standard software applications like word processors, e-mail clients and such. When the product is properly integrated with the IT systems in a given organization, it can help the members of that organization with keeping track of each other's knowledge and expertises. Getting the product properly integrated inside a customer organization is not a simple task, though, and that is why the integration tool that we have developed is a valuable extension of the product.

This chapter describes the Visanti product and its use (Section 7.1) and discusses the challenge of integrating that product with a customer organization's IT systems (Section 7.2). Section 7.3 explains how a domain-specific language can be a solution to such a problem, and Section 7.4 describes how we set out to develop a DSL-based solution. Finally, the post mortem in Section 7.5 wraps up the MORI/SQL project by briefly reviewing how the project developed and what came out of it.

We do not address the actual language and its implementation in the present chapter – this is covered in Chapter 8. Chapter 9 focuses specifically on the formal definition of and the optimizations performed by the compiler we developed.

[1]We shall use the plural form ("we", "our" etc.) throughout this part of the thesis, as if speaking on the behalf of a project team as such, even though the author is solely responsible for the work described and the chapters describing it.

Understanding a problem domain We did not apply a specific method in analyzing the problem domain (such methods are reviewed in [CE00]). Our only instrument for checking the validity of our understanding of the domain was to review the project progress with the domain experts. This approach was not unreasonable as the relevant experts were available to us, and were themselves programmers, fully capable of comprehending our technical work. Also – and this was certainly a complication – our domain was not a stable one: it was defined by a product that kept developing. So none of us (the author or the intended experts) could at any given point be said to understand the domain fully, because it would change *as* we were learning about it.

This has certainly not made it easier to define the problem that we wanted to address with the development of a domain-specific language. But we do hope that after reading this chapter it is possible to follow the design decisions that we present in the next.

The name MORI/SQL MORI is an acronym for "Models of Organizational Resources and their Interactions". The SQL-part of the name signifies that the language is a variant of *SQL*. This relation to *SQL* was not decided on from the beginning of the development project, and we shall occasionally refer to the language and its tools as just MORI, when the relation is not important in the given context.

7.1 The product defining the domain

To understand the domain in which MORI/SQL applies, we must introduce a use case for one of Visanti's applications and what we call *electronic traces of knowledge*. We'll do that by discussing a scenario involving two employees of a fictional *Visanti* customer.

Peter Jensen is employed at the Copenhagen office of EnBank, a larger Danish financial institution. He works in the investment department, providing analyses regarding EnBank's own portfolios as well as advising clients on request.

Peter's day It's Monday morning and Peter arrives at the office around half past eight. As a matter of routine he logs on to see if any e-mails need urgent response before getting a cup of coffee. There is one, a request for a meeting later that day, which he answers before going to get the coffee. In the kitchenette he runs into Tina who wants to hear about the course that Peter attended two weeks before. Both having a busy schedule, they decide talk over lunch. Peter returns to his desk, answers e-mail again, and works on an analysis which is due by the end of the day. A half hour before lunch, a client calls Peter's office mate Michael inquiring about the stock of Aalborg Boldklub A/S, a Danish premier league football club. Peter is the unofficial expert on sports-related stocks so Michael transfers the call to him. Peter answers the question after looking up the club's latest annual report in the company database.

After lunch with Tina, Peter has two meetings and some e-mail before he can go back to his analysis. In the afternoon, Michael returns a favor by answering a call to Peter about a boat manufacturing company. He finishes his analysis around five and goes home.

Peter's electronic trace As a stock analyst, Peter works intensively with the knowledge he has gained through education and work experience. Knowledge is a fickle concept to define and capture. We may perhaps more easily discuss *traces* of Peter's knowledge, in particular electronic traces.

Several of his work functions are mediated by a computer: reading and writing e-mail, looking up information, writing analyses. When performing these functions, his interaction with the relevant tools are observable in a very direct sense of the word. When the electronic *events* making up his interactions are logged we call the logs *electronic traces of Peter's knowledge*. Certainly, we don't know a way to capture his knowledge in itself, but some interactions may point to the fact that he has it. For instance, when he accesses the annual report of the football club, it can be considered as a hint that he may now know things related to that report. Most likely he knew some things before (he had a background that let him understand the report) and very likely he learned something new. Collecting such hints can be useful when trying to find the right employee for a task.

Note that we'll certainly not capture traces of all of Peter's knowledge through logging. Of course, he knows things that he doesn't apply at work. But even knowledge that he *does* apply may not finds its way to an electronic form. He may give and take a lot of good advice (or gossip) over lunch with Tina, and this knowledge may just continue to be exchanged orally. But note also that often such knowledge will eventually affect the way Peter chooses to do his work at the computer and thus produce electronic traces (however indirect and obscure they may be).

Annie's task Annie Hansen also works at EnBank, in their Århus department. She performs credit evaluations related to building projects, specializing mainly in evaluating smaller organizations. On this Wednesday she is looking at the project of extending and modernizing Egerød Stadium. The client is Egerød Boldklub, a second league football club with premier league ambitions but a staff of just fifteen. The club is owned by a fund and two local businessmen.

Annie is not sure about the future financial prospects of the project and feels she needs more knowledge about the branch. Entertainment (and thus sports) is not a core investement area of EnBank, so there are no internal reports available. She looks at the annual reports of some major football clubs but still has some unanswered questions. Annie decides to drag one of the annual reports she just read into the focus area of her *Visanti* application window (see Figure 7.1). It then shows her – among other things – a list of other employees who read the report. Peter Jensen is at the top of the list, indicating that he has consulted the report more often than anyone else in the bank. Of course, he may just have been interested in a few particular details about Aalborg Boldklub A/S. Annie drags Peter's icon into the focus area to obtain a list of other documents that he has consulted significantly more often than others. And indeed, the annual reports of two other football clubs is shown. Annie calls Peter, and finds out the he actually *is* knowledgable about the branch and able to answer her questions. Thus informed, Annie is able to finish her evaluation.

Discussion The *Visanti* application helped Annie solve her problem by letting her browse electronic traces of knowledge collected across offices and platforms all over EnBank. It let her find people and documents relevant to her task at hand. The interface is *relational*: when Annie chooses the relation "Readers of" and picks a document, the application shows a list of related persons, sorted in a relevant order. When she picks Peter and the relation "Has read", she gets back a list of related documents.

There are many knowledge-oriented relations that Annie, Peter, Michael and Tina might want to be able to explore in their work. And while their definitions should be based on the electronic events that make up the traces of knowledge, the interconnection between a relation and the events may be far from trivial. It is this complexity that inspired the development of MORI/SQL.

7.2 The problem of customization

In this section, we discuss in detail the problems related to defining a coupling between knowledge-oriented relations and observable (electronic) events.

While the relations that Annie explored above are rather generic and may be useful in most customer organizations, many other knowledge-based relations may be specifically useful to particular types of setups. Experiences at EnBank might inspire them to define a set of business areas for which the system could assign experts. So in the example, Annie could have chosen an icon for "sports" (or perhaps "entertainment industry") and a relation "experts" to get a list of employees, which would probably have had Peter as its first item. But the list of expertise areas would not likely be suitable for e.g. a phone service provider. They might

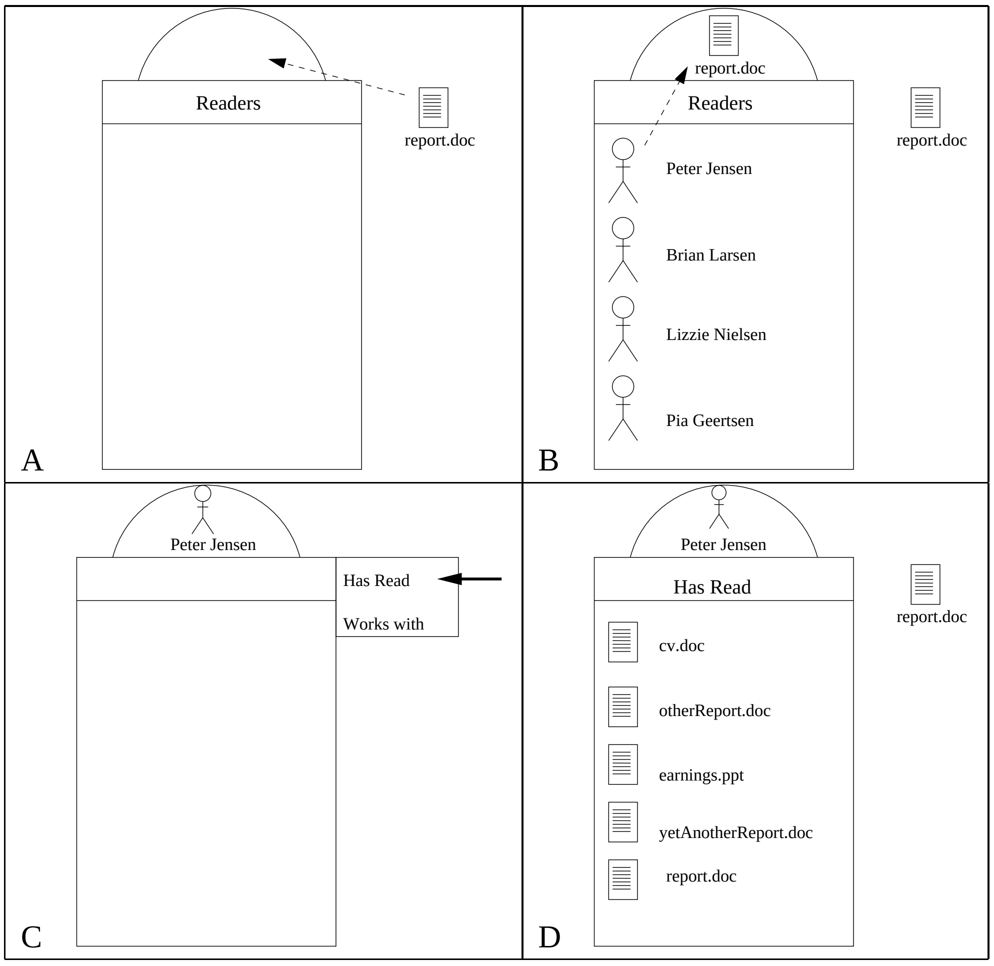


Figure 7.1: Annie's interaction with her *Visanti* tool. The graphical user interface of the tool is just for illustration and somewhat simpler than the real one.

want to be able to explore a relation “user of” which associates a given customer with a list of the services that he uses the most.

But even when the set of needed relations is the same for two organizations, the way these are defined from the observable events may differ. Firstly because logging may behave differently on different platforms. Secondly because working patterns may differ between companies. For instance, if personal use of e-mail is prohibited in one place but not another, e-mail activity is likely to be much different in the two organizations. Thirdly, two customers may use the same relation name about different concepts (e.g. a “client” may be an organization to one of *Visanti*’s customers, but an individual to another).

So for each customer, we need to specify which relations should be explorable and how these should be coupled to the observed events. These definitions must necessarily build on knowledge about the customer’s organization and needs – something we can only hope to obtain through cooperation with customer representatives. The final result of such specifications is code implementing the described system, which is not likely to be readable to the customer representatives. Thus we have a potential gap in the communication between domain experts and programming experts; a gap which poses risks to the customization process.

A customization process To integrate the *Visanti* applications at a customer site, a set of events and relations must be specified and implemented. This process could proceed as follows:

1. Analysis: a *Visanti* systems analyst and a customer representative identify the resource types, resources, events and relations that should be available.
2. Design: on the basis of the analysis, the *Visanti* analyst specifies relations and how these depend on events.
3. Implementation: *Visanti* programmers implement database code to support the specifications.
4. Visualization: *Visanti* GUI programmers implement suitable graphical interfaces to the specified relations.
5. Feedback: The implemented system is presented to the customer who then decides if it matches their expectations.

6-? Revision, optimizations and installation.

This process is problematic in that there are significant potentials for misunderstandings between the involved parties, as indicated in Figure 7.2.

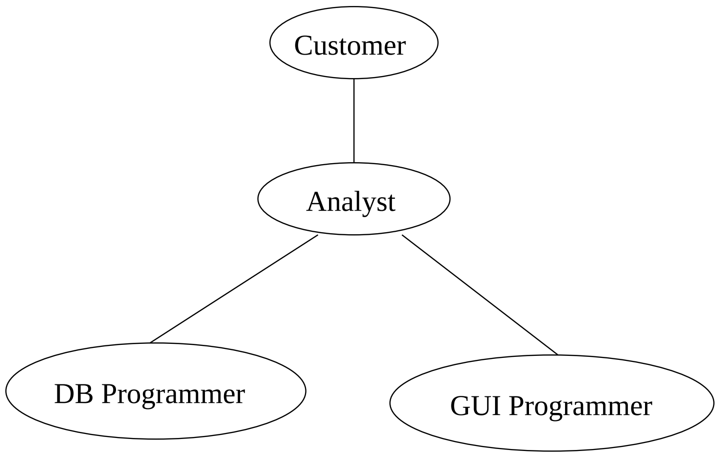


Figure 7.2: Each link of communication between the involved parties constitutes a potential for misunderstandings.

This is serious as an evaluation (revealing misunderstandings) is not realistic until step 5 above. Thus, there is a great risk that a lot of work will be done on the basis of misunderstood specifications.

7.3 A DSL-based solution

The MORI project aimed at addressing the risks described above by introducing a precise model as basis for communication between the involved parties inside *Visanti*.

The first goal was to define a specification language allowing a formal (precise) specification of the results of the analysis in step 1 above. This should minimize the risk of having ambiguities in the proposed model and prevent misunderstandings between the analyst and the programmers. A tool could help building specifications and check that they follow the defined standard.

On top of this system, we would like to put a simulator which, given a specification, lets one test the specified system on a small set of example test data. This should minize the risk of misunderstandings between the customer and the analyst as the feedback loop is shortened significantly (the customer can now evaluate the specification before step 3).

Finally, we would like to build a compiler which, given a specification, performs a significant part of the programming job automatically. This not only minimizes the risk of misunderstandings on the programmer side; the development time should also decrease and thus the time until the customer sees the final product will be smaller.

7.4 The MORI project

We chose an evolutionary project plan. The project was to proceed in a number of iterations, each ending with a review based on an updated progress report consisting of four chapters, documenting progress on the following areas:

1. A catalogue of representative, commented relation examples.
2. A technical report on the formal model of relations.
3. A description of the language, preferably including syntax, formal semantics and user manual.
4. Documentation of the simulator, including overview of the code structure and example input and output.

Each report should also have a separate section on detailed goals for the next iteration. Given the somewhat unclear (and unstable) statement of the problem domain it seemed wisest to develop the four above goals as much in parallel as possible.

Risks We anticipated several potential risks – inherent in any industrial/academic project – to the project plan:

- Unstable requirements. (The needs of *Visanti* may change too quickly).
- Unstable design. (The design of the system that we model may change).
- Insufficient communication between research and development.

All of these had caused problems in the earlier stages of the Ph.D. project. We don't see any foolproof solutions but suggest that the issues are addressed by having periodic project reviews. These reviews should serve as workshops for exchanging information and ideas between research and development. Each of our reviews was attended by the project responsible (the author), the supervisor from *Visanti* and at least one relevant developer.

Further risks included

- Potential problems with interfacing our code with other *Visanti* code. (This is always relevant when developing at separate sites).
- The practical utility of the project results. (Is it useful/efficient enough?)

We had no clear strategy to address the former. We intended to integrate all prototypes at the earliest possible stage to avoid such problems.

The latter risk really constituted an open question at the time of the project launch. We agreed to try to keep an open eye for potential drawbacks of our approach (the most obvious of which would be overkill[2]) as well as potential benefits (beyond the ones the project was intended to achieve). A candidate benefit was ease of porting query implementations to new platforms or revised database schemes. Updating a compiler provides a uniform way of updating the queries it compiles.

7.5 Post mortem of the MORI project

The MORI project is finished. We end this chapter with a brief resume of its evolution and results.

What was actually delivered? We designed a language for specifying relations based on knowledge traces, as discussed in Section 7.3. We also implemented a highly optimizing compiler for the language and integrated it with the database component of the Visanti product.

Because of organizational limitations requirement specifications relating to the customization process turned out to be hard to come by. To handle that problem we chose to focus on the the people we *did* have access to: the database programmers. Keeping the long-term goals in mind, we closely inspected the existing implementations of the relations and discussed the design space with the programmers. We aimed at simplifying their tasks and making the code accessible to others.

Severe time restrictions also put limits to our ambitions. In the end we chose to only implement the compiler, realizing that this would be useful to the database programmers and still helpful to the other groups shown in Figure 7.2.

What actually happened? The planned iterations were executed on time, although the status reports never quite reached the state of completion we had hoped for. They were still good enough to serve their main purpose of preparing the project reviewers.

At the end of the last iteration a compiler was delivered. It did the intended job but had many opportunities for expansion and improvement. Furthermore, one of the anticipated risks hit the project: the requirements to the output code had changed.

[2]Or “shooting sparrows with cannons”, as a Danish idiom goes.

Still, we found the results to be promising and decided to go ahead with more iterations to update the compiler to the new output code format and to add more functionality to the input language. During this period another – unanticipated – risk surfaced, when the project leader left the organization. This did not ease the process of integrating the compiler project into the rest of the system, but a final version of the compiler was released on schedule and is now integrated in the Visanti product.

Evaluating the project results The explicit goal of the project was to develop a tool that could improve on the process of customizing Visanti's system at customer sites. It is our thesis that this goal has been achieved by the tool we delivered.

To evaluate whether this thesis is correct, we must analyse the execution of several customization projects, some of these performed using our tool, and others not using our tool. We do not see how the thesis could realistically be tested by theoretical arguments or by controlled experiments, as we cannot describe with accuracy what would be a representative customization project. The thesis must be tested in the chaotic world of real customer environments.

We have not had access to tracks of customization projects, either using or not using the MORI tool, and it does not appear that such information will be available in the near future, before the deadline of this thesis. Thus, a scientific evaluation the project results cannot be performed. It is our hope that the following chapters, by demonstrating the expressiveness of the language and the services of its compiler, will convince the reader of the value of MORI/SQL. As a final, unscientific note we might add that our main user group, the database programmers of Visanti, have expressed a very positive attitude towards our results.

8 - Design of the MORI tool

In this chapter we describe the overall design of the MORI tool, including the external requirements to the solution as well as the choices we made, and their rationales.

The tool is meant to fit into an existing (and evolving) system and we cannot ignore the requirements stemming from this dependency: the code we write and the code *it* produces must interface with the rest of the system's code. We should also focus on solving problems that are not handled by other parts of the system, otherwise the tool might be considered superfluous. Finally, our tool should become an integral part of the system's code base and therefore not remain in the "ownership" of the author. Other (industrial) developers must eventually take over responsibility of upgrades and bug fixes to the tool. So not only should it be reasonably well-documented, the tool must also be designed and implemented in a way that the developers can understand without studying many technical issues in semantics and programming language theory.

Thus we begin the chapter by discussing the overall layout of the system that our DSL should be part of (Section 8.1). The last part of that section explains our choices regarding the general structure of the MORI tool. The following section dicusses what functionalities MORI/SQL applications could ideally have, and explains our choices regarding the language itself and its syntax. Section 8.3 outlines some of the optimizations that we needed the MORI tool to perform. This is important as efficiency was central to our application. Chapter 9 is devoted to an especially complex part of the optimization process. Sections 8.4 and 8.5 describes the implementation of MORI/SQL in detail. We conclude in Section 8.6.

8.1 The context of MORI

In this section we describe part of the Visanti product that the MORI tool relates to. The first subsection the actual database that stores the electronic tracks of knowledge, called the *Interaction Log Server* (ILS). The second subsection shows how queries are sent to that database, and points out where the MORI tool fits in. It also discusses issues relating to implementation style and implementation languages.

8.1.1 ILS: an interaction log server

In this section we describe the Interaction Log Server. What we describe is not the real system architecture but rather a distilled version that brings up all the elements that we will have to consider, while ignoring other parts of the system that are not relevant to our task.

As its name implies, the ILS registers (logs) *interaction events*, that is to say, events that are the direct result of a human user interacting with some software application. In other words, interaction events describe observable human behaviour and not, say, automatic behaviour of networked servers communicating.

Interaction events come in many flavours: editing a document, sending an e-mail, posting a message on the internal intranet – all these interactions cause interaction events. We consider these "flavours" as atoms (i.e. unstructured identifiers called e.g. `edit-doc`, `sent-mail` and `msg-post`) making up a set of such flavours. This set will not be the same in all installations of the ILS, but in any installation it will be a finite set of identifiers.

When an interaction event is registered, the ILS naturally stores the flavour of the event as well as its time stamp (when the event took place). It also stores a *subjectID* and an *objectID* of the event. The subject will usually be an ID of the user that made the interaction happen. The object will be an ID of whatever or whoever the event "happened to". In the case of an `edit-doc` event the object would be a unique identifier of the document that was edited. In the case of an `sent-mail` event the object is an ID of the receiving user. The ILS keeps and updates a set of identifiers, having a unique key for any document and user (and perhaps other entities) that is used in the registration of an event. As mentioned, the set of flavours is chosen when the ILS is installed. With each flavour there is a definition of what constitutes subject and object. There is also a definition of event *size*. The size is an integer that is also stored on each event. The measure may of course be very different between different event flavours. For e.g. an `edit-doc` event, the size may be the number of bytes added to or removed from the document.

The registred interaction events are stored in a relational database table with the schema *event(flavour,subjectID,objectID,timestamp,size)*. As mentioned above, *flavour*, *subjectID* and *objectID* are considered atomic. They all have type integer in the actual representation. So does *size* (although the size should never be negative). Values of field *timestamp* are of type `date` (a suitable type implemented by the database).

An example *event* table is shown in Figure 8.1.

flavour	*subjectID*	*objectID*	*timestamp*	*size*
`sent-mail`	32	15	08:31	12
`sent-mail`	32	40	08:44	50
`sent-mail`	32	20	08:49	20
`edit-doc`	32	2566	08:55	200
`read-doc`	32	9804	08:58	154
`sent-mail`	40	8	09:01	12
`sent-mail`	8	40	09:10	22
`sent-mail`	40	32	09:15	77
`sent-mail`	32	40	09:22	1
`sent-mail`	31	20	09:30	35
`sent-mail`	31	8	09:40	15
...	...	...	...	...

Figure 8.1: An example of what rows in the *event* table might look like.

8.1.2 Querying the ILS

Figure 8.2 shows part of the Visanti system; the part that our tool should work in the context of. At the top is a layer of code implementing remote procedure calls. This is the code that a client application contacts when it needs to perform a query to the *ILS*.

A call to the RPC layer is site-specific: its meaning depends on the local setup of the ILS as discussed in Section 8.1.1. An underlying layer decodes the site-specific request into the name of an SQL query function and a list of actual parameters for that function. We dub this layer the *site-specific configuration layer*, or just the *SSC* layer. Having determined which function to call and which parameters to pass to that function, the call is performed. This results in an SQL query being executed on the ILS database. The results of this query is then returned up through the layers and sent to the requesting client. The existing components are coded in Java, except the query functions which are in PL/SQL, a vendor-specific version of SQL.

Our main goal was to provide a tool that will allow query functions to be specified in MORI/SQL, as shown with the dotted lines in Figure 8.2. Each MORI/SQL relation must be compiled into a PL/SQL function, simply because this is the only language at hand that we can access the *ILS* with. Interpretation in PL/SQL would be possible but very awkward and inefficient. Partial evaluation might compensate for some of the inefficiency, provided we developed a partial evaluator for PL/SQL, but most of the speedup had to be achieved by writing a query optimizer (this is the topic of Chapter 9), and that task is one that PL/SQL is *not* well suited

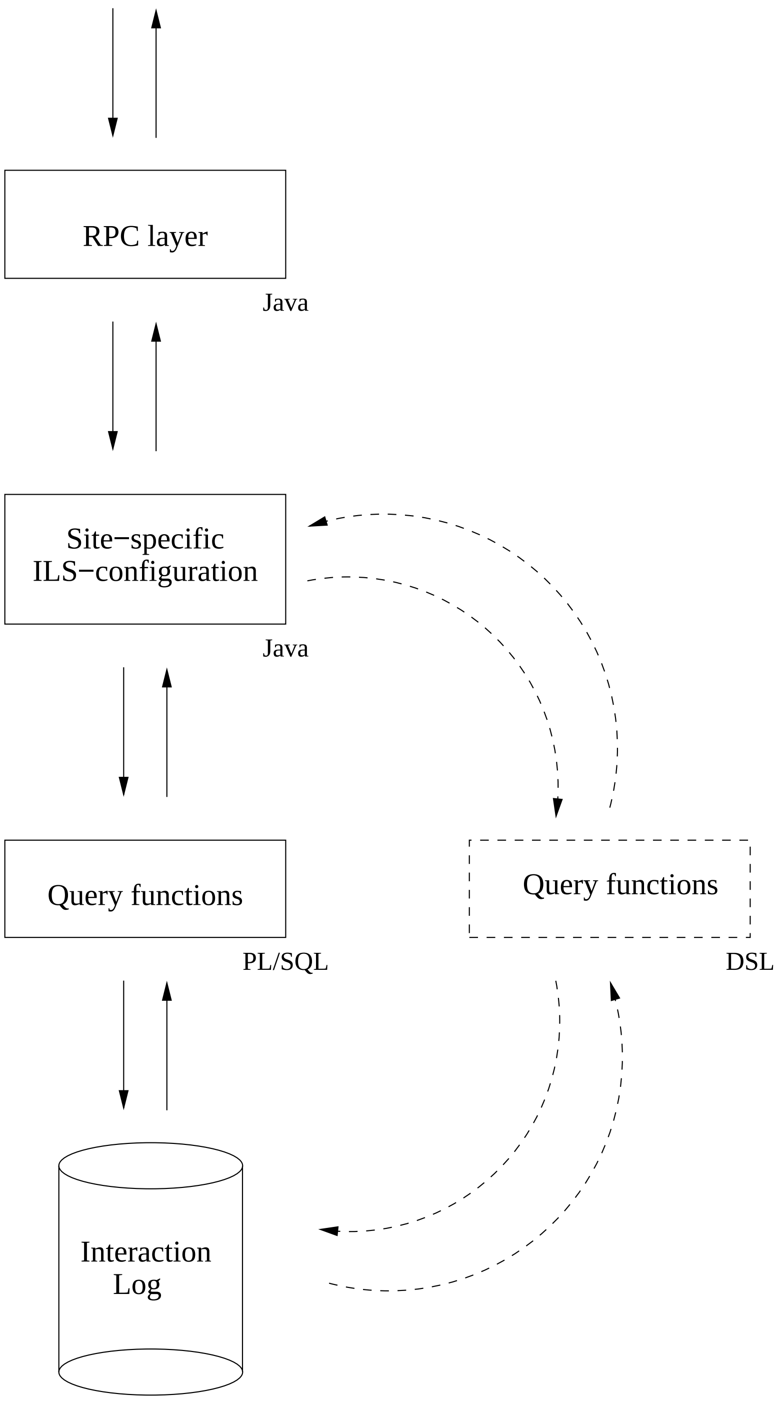


Figure 8.2: Graphical overview of the context in which our DSL should fit. The modules and arrows demarked by full lines are already in existence, the module and arrows demarked by the punctured lines is the part that we are designing. We have not shown the part of the system that updates the interaction log.

for.

So we chose to write a compiler from MORI/SQL to PL/SQL. The compiler was developed in *Java* for three reasons:

- This would interface smoothly with the rest of the system.
- The industrial developers associated with the project were of course very familiar with that platform.
- We did not want to introduce too many third-party products (e.g. a *Haskell* compiler/interpreter as in [LM99]) into the system, for fear of complicating the integration task.

8.2 Example relations and a language for their specification

We now move on to our primary purpose: modelling knowledge-based relations. The first subsection explores the design space of relevant relations by giving a list of practical examples. Section 8.2.2 then describes the main choices that we made towards the definition of MORI/SQL. Finally, Section 8.2.3 presents the syntax and informal semantics of the language that we settled on.

8.2.1 A list of example relations

In chapter 7 we discussed relations abstractly, with few examples. In this section we give a number of concrete examples that we used to analyze the design space of relations. These examples come from practice and are not arranged to fit any reasonable pattern. Indeed we found that we could not accommodate all of them in our domain-specific language and chose to model some of them. This will be discussed in Section 8.2.2.

As we saw in the product use case, illustrated in Figure 7.1, the system always responds when a relation *and* a resource is specified. We call this input (a relation coupled with a resource) a *query*. The result of a query – the list of related resources with associated information – is dubbed a *table*.

MostEdited With this query we wish to list for a given document d those users who caused most `edit-doc` events to d before a given date t. In other words the resulting table should tell us who was the most active editors of d up to time t. The table should be sorted having the most active first, and with each user the number of relevant `edit-doc` events and the date of the last such event (before t) should be stated. An example result table is shown in Figure 8.3.

objectID	NumEvents	LastDate
2	8	12/5-2002,14:15
7	6	11/5-2002,16:21
21	6	11/5-2002,16:19
9	5	13/5-2002,08:24
...	...	...

Figure 8.3: Example result of a *MostEdited* query.

DaysBefore	NumReads
0	224
1	48
2	63
...	...
30	0
31	0

Figure 8.4: Example result of a *DocReadActivity* query.

Note that order is really important. The point of the query is to reveal the most active editors. Thus, the main objective is to show the highest-ranking results (rather than return the exact number of qualifying result records, for instance). This theme – often called relevance sorting – is common in information retrieval and it will apply to many other examples.

DocReadActivity The result of this query on a document d should be the distribution of `read-doc` events related to the given document d. The distribution should be on a per-day basis for 31 consecutive days up to a given date t. I.e. "How many times was d read in the days up to t?". A result table could look like the one in Figure 8.4. Here the first dimension (*DaysBefore*) is of fixed size – we always consider 31 days – and the natural order on the records is of course the chronological order.

ReadersAlsoRead Given a document d we wish to know which other documents were read by the readers of d. Order is certainly important here – we choose to give priority to those documents that were read by the largest fraction of d's readers. Other choices of priority are of course possible. An example result is shown in Figure 8.5.

An even more useful query would select those documents that are often read

DocId	Fraction
199	99%
535	98%
164	35%
...	...

Figure 8.5: Example result of a *ReadersAlsoRead* query.

UserId
47
12
101
...

Figure 8.6: Example result of a *CollaboratesWith* query.

by *d*'s readers *but seldom by others*. This would assign low priority to useless positives like the corporate intranet homepage, which may be read very often by everyone.

CollaboratesWith This is an intricate relation. How do we figure out (from logged interactions) who collaborates with a given person *X*? A number of possible clues spring to mind:

- If *X* and *Y* exchange emails often they may be collaborating.
- If *X* and *Y* edit the same documents (i.e. co-author documents) they very likely collaborate.
- If *X* often prints *Y*'s documents and vice versa, they may be collaborating.
- If *X* and *Y* are assigned to the same project, we must assume that they collaborate.

The precise definition of this query seems like it might depend on the given system (which events do we monitor?) and the given organization's usual working patterns. A result table might look like the one in Figure 8.6.

8.2.2 Choices regarding DSL content and syntax

Choice #1: Not to replace the SSC layer This choice was implicit in Figure 8.2. As mentioned above the SSC layer binds flavour parameters to queries, so that a PL/SQL *Most* function can implement e.g. *MostEdited*, *MostRead*, *Most-Mailedto* and many more queries. The first version of MORI offered the same functionality and replaced both the SSC layer and the PL/SQL code for the queries it supported. Given that it could not support each and every function in the PL/SQL code, and thus not replace the SSC layer entirely, we ended up with two ways of implementing the same functionality. This caused a lot of problems in integration. As the SSC layer was already integrated into the system and used by many people, we chose to remove the facility from MORI.

Choice #2: SQL-like syntax We put up several suggesions for the general syntactic style of the language. The first suggestion was a domain-specific variant of *Prolog*, while the second was based on symbolic logic. The development team's internal communication turned out to become smoother when our program examples were presented as a domain-specific variant of SQL. This version was much more accessible to everyone (but the author, who liked the different suggestions equally). Discussing who would be the main target group of the language, we agreed that SQL was the language most likely to be familiar to these users, and we adopted that style.

Choice #3: Which examples to focus on Some of our example relations (not given above) were dropped after a some consideration. Some examples represented a set of queries that involve other parts of the system than the ones shown in Figure 8.2, subsystems that perform textual analysis of documents. We dropped these queries to limit our task.

Of the four remaining examples (given in Section 8.2.1), we found it easy to fit the three (*MostEdited*, *ReadersAlsoRead* and *CollaboratesWith*) into a logical pattern. We'll skip the details of this pattern – it was in essence equivalent to the algebra presented in Chapter 9. But *DocReadActivity* did not fit and we discussed whether the *Activity* queries had an interesting span of different definitions. Concluding that it did not, we also dropped that example and focussed on the three mentioned above.

8.2.3 The final syntax of MORI/SQL

Through writing a number of examples and making other (minor) choices, we ended up with the grammar shown in Figure 8.7. An example MORI/SQL relation is given in Figure 8.8.

A relation is described as an SQL query parametrized by a *focus resource* and one or more flavour paramters. The focus resource is the resource (e.g. a user or a document) that is being related to other resources. It is an implicit parameter of

relation	::=	RELATION *Id* **(** parlist **)**
		BEGIN
		query
		ORDER BY *Id* (ASC | DESC)
		END
parlist	::=	*Id* IN EVENT (, *Id* IN EVENT)*
query	::=	select from where groupby
select	::=	SELECT *Id* AS *Id* **,** explist
explist	::=	exp AS *Id* (, exp AS *Id*)*
exp	::=	MAX *Id*
	|	SUM *Id*
	|	COUNT *****
	|	**(** exp **)** ***** **(** exp **)**
	|	**(** exp **)** / **(** exp **)**
from	::=	FROM fromlist
fromlist	::=	*Id Id* (*Id Id*)*
where	::=	WHERE wherelist
wherelist	::=	constraint (AND constraint)*
constraint	::=	FOCUS = *Id*
	|	*Id* = FOCUS
	|	FOCUS $<>$ *Id*
	|	*Id* $<>$ FOCUS
	|	*Id* = *Id*
groupby	::=	GROUP BY *Id*

Figure 8.7: Grammar of MORI/SQL. Asterisks and parantheses are terminals, except for the parlist, explist and wherelist productions.

```
RELATION most(chosen_flavour IN EVENT)
BEGIN
  SELECT ev.to AS output, SUM ev.score AS score
  FROM chosen_flavour ev
  WHERE ev.from = FOCUS
  GROUP BY output
  ORDER BY score DESC
END
```

Figure 8.8: An example MORI/SQL relation.

every relation. The `EVENT` keyword denotes a domain-specific type for flavours. In MORI/SQL, each flavour is presented to the user as a separate database table containing the records with that flavour.

A query writer may join one of these flavour tables several times by letting it appear several times (with different aliases) in the `FROM` declaration. The joined tables can be restricted by any number of constraints in the `WHERE` declaration. Each constraint defines a criterion that records must satsify. The criterion is one of the following:

- That a given field must match the focus resource.
- That a given field must be different from the focus resource.
- That two given fields must be identical.

The records that satisfy all criteria are then subjected to a `GROUP BY` operation. The field in the `GROUP BY` declaration must be of resource type, not integer or time stamp types. This field is the output resource (it lists the resources that the focus resource is related to) and it must also be the one following the `SELECT` declaration.

When grouping a table T by one of its fields, the records in T that agree on that field are all "smashed" into just one record. One can select any positive number of expressions that aggregate (e.g. sum up) fields in T. If table T' is the subset of records of T that all have value v in the `GROUP BY` field, the one record that will represent T' after grouping will contain e.g. sums of columns in T'. The result after grouping is sorted after one of its integer or time stamp fields.

We also refer the reader to the formal semantics of MORI algebra in Chapter 9.

8.3 Structure of the target code

In this section we study the PL/SQL code that the output of our compiler should be able to replace. This is important as the code is heavily optimized in ways that out code generator must imitate.

If we look at the *MostEdited* example in Section 8.2.1, we might expect the code for this query to have an overall structure like in Figure 8.9. The input parameter *focus_resource* represents the ID of the document d that is in focus.

The truth is more complicated, however. The functionality for *MostEdited* is implemented by a function like the one in Figure 8.10. The *Most* function generalizes over flavours. We get the *MostEdited* query by binding the *chosen_flavour* parameter, not to 4 (representing the `edit-doc` flavour) but to a number that represents the *inverse* of `edit-doc`, where *subjectID* and *objectID* are swapped in

```
FUNCTION MostEdited(focus_resource IN INT)
RETURN  [some type representing a table]
BEGIN
  SELECT subjectID,
         SUM(size) AS numevents,
         MAX(timestamp) AS lastdate
  FROM ils
  WHERE flavour = 4  //where 4 represents 'edit-doc'
    AND objectID = focus_resource
  GROUP BY subjectID
  ORDER BY score
END
```

Figure 8.9: The *MostEdited* query as we might expect it to look. All-capitals words are PL/SQL keywords.

each record.[1] This binding is performed by the site-specific configuration layer shown in Figure 8.2. The many extra parameters are either sent by the client or provided by the site-specific configuration layer as well. The real *Most* function contains seven subqueries of which four are tables (one of these is of course the ILS table). It is 43 lines long.

Templates Or rather, it *would* be 43 lines long, if it wasn't for the further optimizations. To make the query perform adequately on the huge amount of data that the ILS (and other tables) usually contains, the SQL query shown above is actually written as a formatted string template, see Figure 8.11.

Note the single quotes and the colons. The string constant that is constructed is formatted by substituting the elements starting with colons by the values in the list after `USING`. There are several hundred of these, and their order in the list must be exactly the same as their use in the query. Needless to say, writing queries this way is tedious and error-prone. But it produces very fast queries.

The *real* (honest!) function is about 115 lines long and not fit for the untrained eye.

[1]The reason for this inversion is technical and related to database optimization.

```
FUNCTION Most(focus_resource IN INT,
              chosen_flavour IN INT,
              //many parameters specifying limits on
              //security restrictions on viewable files,
              //groups that output resources should belong to
              //and number of results to return.
             )
RETURN  [some type representing a table]
BEGIN
  SELECT objectID,
         numevents,
         lastdate,
         //more info found in other tables
  FROM //other tables and
    SELECT objectID,
           SUM(size) AS numevents,
           MAX(timestamp) AS lastdate
    FROM ils
    WHERE flavour = chosen_flavour
      AND subjectID = focus_resource
      AND //constraints controlled by the
          //extra parameters
    GROUP BY objectID
    ORDER BY score
  WHERE //further constraints
END
```

Figure 8.10: The general *Most* function with many parameters restricting the query.

```
FUNCTION Most(focus_resource IN INT,
              chosen_flavour IN INT,
              //many parameters specifying limits on
              //security restrictions on viewable files,
              //groups that output resources should belong to
              //and number of results to return.
             )
RETURN  [some type representing a table]
BEGIN
  //Some lines caching time-critical data
' SELECT objectID,
         numevents,
         lastdate,
         //more info found in other tables
  FROM //other tables and
    SELECT objectID,
           SUM(size) AS numevents,
           MAX(timestamp) AS lastdate
    FROM ils
    WHERE flavour = :chosen_flavour
      AND subjectID = :focus_resource
      AND //constraints controlled by the
          //extra parameters
    GROUP BY objectID
    ORDER BY score
  WHERE //further constraints
' USING
  //Here is a list of 'dynamic' values
  //including the cached ones and the input parameters.
END
```

Figure 8.11: The *Most* query on a template form.

$$\boxed{\begin{aligned} \text{MORI/SQL} &\xrightarrow{\textit{parser}} \text{MORI algebra}\\ &\xrightarrow{\textit{optimizer}} \text{MORI algebra}\\ &\xrightarrow{\textit{codegen}} \text{XML}\\ &\xrightarrow{\textit{XML parser}} \text{PL/SQL} \end{aligned}}$$

Figure 8.12: The transformational approach we chose for implementing MORI/SQL. See also Figure 8.13.

8.4 Overall structure of the implementation

The implementation task was to implement a compiler from MORI/SQL to PL/SQL in *Java*. After some prototyping we decided on a transformational approach as illustrated in Figure 8.12. The source MORI/SQL code is transformed through four steps into the target PL/SQL code:

1. The first step is implemented by the parser. It transforms a MORI/SQL relation into a term in the MORI algebra we present in Chapter 9. This term is very close to simply being an abstract syntax tree for the relation.

2. The term is then rearranged into one that has the same semantics but is expected to yield a more efficient query than the original. This transformation is performed by the *optimizer*, which is also discussed in in Chapter 9.

3. From the optimized term, the system generates target code. For each MORI/SQL relation, the target code is a PL/SQL function that resembles the example in Figure 8.10, although it is embedded in XML ([HM02]).

4. The purpose of embedding each PL/SQL function in XML is to avoid the complications related to the template format shown in Figure 8.11 during code generation. In the last step of the compilation process, an XML parser uses the markup to *lift* each PL/SQL function into the template format.

We found this separation of the compilation process useful because many simple changes can be implemented without understanding all of the steps in detail. This was important as we needed the domain experts – database programmers – to update the code generation steps without necessarily studying the details of optimization. The original design had even more steps in the compilation process, mainly because we anticipated the need for several domain-specific languages to be compiled into PL/SQL, and we wanted to be able to share large parts of the code between the different compilers.

Integration of MORI/SQL and PL/SQL In the final implementation we *did* use one of the steps for another purpose. The ability of our XML parser to lift PL/SQL code into template form, i.e. to transform Figure 8.10 into Figure 8.11, was just what the database programmers needed for writing those PL/SQL functions that are not modelled by MORI/SQL. Thus we made the interface of the XML parser available to the database programmers.

Having done this, it seemed natural to use the XML format to integrate PL/SQL and MORI/SQL fully. The final format allows the programmer to write both PL/SQL functions and MORI/SQL relations (a tag surrounding the code signals whether it is one or the other). The XML parser cuts out each MORI/SQL relation, delivers it to the MORI/SQL compiler, and pastes the output PL/SQL back into the XML document, from where it will finally be lifted to template form. The process is illustrated in Figure 8.13 and explained in more detail in Section 8.5.

The following paragraphs provide some notes on the first three steps of the compilation process.

Parsing, optimization and code generation The parser reads MORI/SQL input and constructs an algebraic term. The term is represented as a Java object of type `RelAlg.Query`, a proprietary class written by the author. The parser's functionality can be described by a simple denotational semantics. This will be given in Chapter 9 (in Figure 9.2) when we have defined the algebra that the terms belong to. The parser itself was written using the *Cup* parser generator for *Java* (cfr. [App98]). Employing the use of this third-party application was deemed necessary.

The optimizer produces a term that is functionally equivalent to (but expect to yield faster code than) the one constructed by the parser. This optimization is quite complex and we devote Chapter 9 to describing the principles behind it.

The optimizer outputs code in the style of Figure 8.9. The *codegen* part of the compiler adds all the aspects present in the code sketched in Figure 8.10. The final output is PL/SQL embedded in XML.

8.5 The XML parser

As explained at the bottom of Section 8.4, the XML parser ended up being used for three purposes:

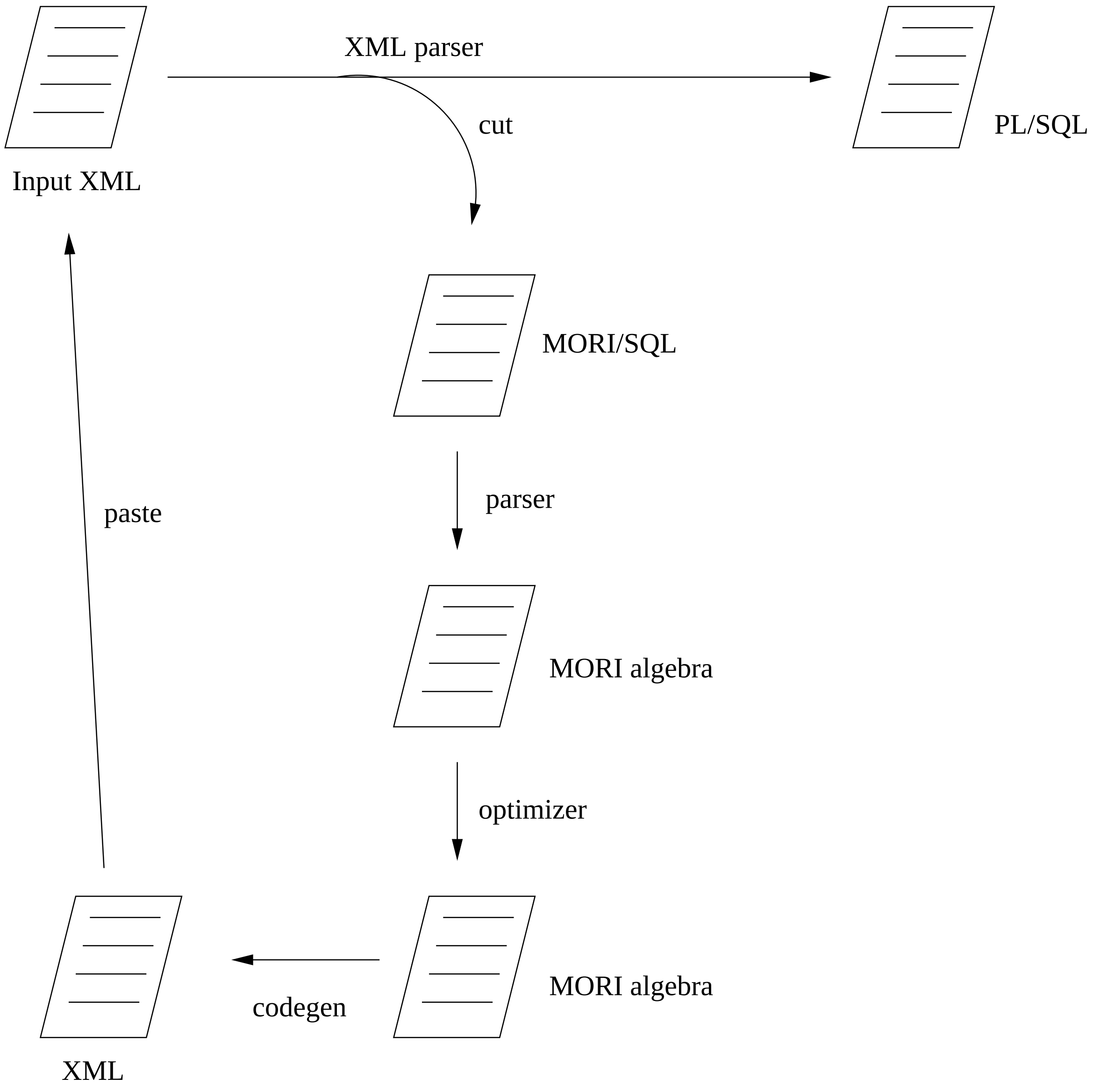


Figure 8.13: The whole process of transforming XML input.

- Inside the MORI/SQL compiler, for lifting the target code to the template style of Figure 8.11.
- Independent of the MORI/SQL compiler, for formatting ordinary PL/SQL functions in the same template style.
- As an outermost encapsulation of the MORI/SQL compiler, allowing a programmer to write both PL/SQL and MORI/SQL in one file.

The input to the XML parser must obey the Document Type Definition in Figure 8.15. An example input file is given in Figure 8.14. The root tag is `<pl-mori>` and has an associated version number, indicating the version of the format used in the file. The content of the `<pl-mori>` tag is PL/SQL code.

Interaction between the XML parser and the MORI/SQL compiler Any MORI/SQL code must be enclosed in a `<mori-sql>` tag. When the parser sees a `<mori-sql>` tag, it passes the content to the MORI/SQL compiler. The compiler performs the first three steps of the compilation process and returns the resulting code (recall that the third step yields XML-embedded PL/SQL functions). The code replaces the `<mori-sql>` tag, and is then processed as it would have been if it had been inserted literally in the file. For instance, the MORI/SQL relation *most* in Figure 8.14 would be replaced by a PL/SQL function the body of which is identical to the body of the *OtherMost* function appearing just below it.

Lifting the PL/SQL functions The net result of this process is that the input file in effect contains PL/SQL functions embedded in XML. What the XML parser must do with these functions is to *lift* each into the template style we demonstrated in Figure 8.11.

The PL/SQL `USING` facility is akin to the functionality of `printf()` in *C*. The essence of the lifting is that

- the actual SQL query must be put inside single quotes, and
- each *dynamic* value occurring in the query must be replaced by a placeholder and instead be put in a value list following the query.

In this context a *dynamic* value is either an actual parameter of the PL/SQL function or a value computed in the PL/SQL statments preceding the actual SQL query. The order in which the values appear in the list is important, as these are matched left to right with the placeholders appearing in the quoted SQL query.

A complication arises from the fact that sometimes a dynamic value is not scalar, but an array like *clearances* in Figure 8.14. That particular array calls for 150 placeholders to be placed in the quoted SQL query and correspondingly 150 dynamic values to be listed after `USING`. There are also special record values called *attributes* that need special handling, but they are of no general interest and we shall not discuss them here.

Concrete syntax The `<using-query>` tag demarks the actual SQL query. When scanning the content of that tag, the XML parser outputs all ordinary text literally but replaces any dynamic value with a placeholder and stores it in an internal list. A scalar dynamic value is represented by the `<plug>` tag. Each occurence of a dynamic array is represented by an `<array>` tag. The `<plug>` tag has one attribute that is assigned the actual dynamic value. An `<array>` tag is a reference to an earlier `<define-array>` tag, which declares how the given array is to be accessed. We shall not discuss the details of the `<define-array>` tag.

The XML parser is implemented as a *SAX* parser ([HM02]). A Document Type Definition of valid input for our XML parser is given in Figure 8.15. The `<attrlist>` and `<attr>` tags are used to represent the above-mentioned attribute values.

8.6 Conclusion

The MORI project has resulted in the definition of a domain-specific language, called MORI/SQL, and a compiler that translates queries in MORI/SQL into highly optimized queries in regular SQL. Use of MORI/SQL (rather than regular SQL) relieves the programmer of many worries that the production of efficient queries in the *Visanti* would otherwise cause. MORI/SQL makes the definition of complex queries possible, simple even.

This chapter has explained the main design choices behind the MORI/SQL language, its syntax and its compiler. The chapter also described several crucial features of the MORI/SQL compiler's implementation.

```
<?xml version="1.0"?>
<pl-mori version="2.0">

<mori-sql>
RELATION most(chosen_flavour IN EVENT)
BEGIN
  SELECT ev.to AS output, SUM ev.score AS score
  FROM chosen_flavour ev
  WHERE ev.from = FOCUS
  GROUP BY output
  ORDER BY score DESC
END
</mori-sql>

function Other_Most(focus_resource IN INT,
                    chosen_flavour IN INT,
                    user_class     IN INT)
RETURN TABLE_TYPE IS
  rc TABLE_TYPE;
  clerances INT_TABLE;
BEGIN
<define-array name="clearances" size="150" call="ReadClElems"/>
  //This fcn call returns an array with 150 elements:
  clearances := GetClearances(user_class);
  OPEN rc FOR
<using-query>
    SELECT objectID AS output , SUM(score) AS score
    FROM ils
    WHERE ils.subjectID = <plug value="focus_resource"/> AND
          ils.classification IN <array name="clearances"/> AND
          ils.flavour = <plug value="chosen_flavour"/>
    GROUP BY output
    ORDER BY score DESC
</using-query>
    RETURN rc;
END;
</pl-mori>
```

Figure 8.14: An example input file to our XML parser.

```
<!DOCTYPE pl-mori [
  <!ELEMENT pl-mori (#PCDATA | define-array | using-query | mori-sql)*>
  <!ATTLIST pl-mori version NMTOKEN #REQUIRED>
  <!ELEMENT define-array EMPTY>
  <!ATTLIST define-array name NMTOKEN #REQUIRED>
  <!ATTLIST define-array size NMTOKEN #REQUIRED>
  <!ATTLIST define-array call CDATA #REQUIRED>
  <!ELEMENT using-query (#PCDATA | array | plug | attrlist)*>
  <!ELEMENT array EMPTY>
  <!ATTLIST array name NMTOKEN #REQUIRED>
  <!ELEMENT plug EMPTY>
  <!ATTLIST plug value NMTOKEN #REQUIRED>
  <!ELEMENT attrlist (attr)+>
  <!ELEMENT attr EMPTY>
  <!ATTLIST attr name CDATA #REQUIRED>
  <!ATTLIST attr type (number | string | date) #REQUIRED>
  <!ATTLIST attr field CDATA #REQUIRED>
  <!ELEMENT mori-sql (#PCDATA)>
]>
```

Figure 8.15: Document Type Definition describing valid input to our XML parser.

9 - A domain-specific relational algebra

It's the nexus of the crisis
the origin of storms
Just the place to hopelessly
encounter time
and then came me!
"Astronomy", Blue Öyster Cult

What we present in this chapter is a domain-specific version of relational algebra (see e.g. [Die01]). The algebra (called *MORI algebra*) is an intermediate language in the compiler from MORI/SQL to PL/SQL. The point of defining this domain-specific algebra was that terms in the algebra

- are straightforward to generate from MORI/SQL input,
- are easy to generate PL/SQL code for, and
- can be optimized using automatic transformations.

The first objective is treated in Section 9.2. The second objective was not particularly hard to obtain, and we shall not discuss it here. The third objective is treated in detail in Section 9.5. An important part of the optimization is a generalization of the *Generalized Coalescing Grouping* principle outlined by Chaudhuri and Shim in [CS94]. To our knowledge, Section 9.5.2 provides the first proof of the correctness of those particular optimizations based on formal semantics.

The rest of the chapter is devoted to the theory needed to address the objectives: syntax, semantics and types for the MORI algebra. We conclude in Section 9.6.

9.1 Syntax

The syntax of the MORI algebra is shown in Figure 9.1. We do not expect the reader to have an intuitive idea about what these operators mean. You can think of each variable as a table in a database. A query Q (parametrized by a "binding" value r) will correspond to a view in that database. In relational algebra, σ-operators are restrictions i.e. operations that remove rows from the result, while

$$
\begin{array}{rlcll}
Q, Q_1, \ldots, Q_n \in & \mathit{Query} & ::= & x(c) & \\
 & & | & Q_1 \times \cdots \times Q_n & ; n > 0 \\
 & & | & \sigma_{bind(c)}\, Q & \\
 & & | & \sigma_{bind\neg(c)}\, Q & \\
 & & | & \sigma_{eq(c_1 \cdots c_n)}\, Q & ; n > 1 \\
 & & | & \Pi_{D/c'=c}\, Q & \\
D \in & \mathit{Defs} & ::= & c_1 = E_1 \cdots c_n = E_n & ; n \geq 0 \\
E_1, \ldots, E_n \in & \mathit{Expr} & ::= & \mathit{max}\ c & \\
 & & | & \mathit{sum}\ c_1 \cdots c_n & ; n \geq 0 \\
 & & | & E_1 * E_2 & \\
 & & | & E_1 / E_2 & \\
x \in & \mathit{Variables} & & & \\
c, c', c_1, \ldots, c_n \in & \mathit{Identifiers} & & &
\end{array}
$$

Figure 9.1: The syntax of MORI algebra.

Π-operators are projections: operations that remove and introduce columns in the result. Product is Cartesian product, sometimes called the Cartesian join. We have deliberately chosen a small set of expressions to keep things simple in this presentation.

The operation named *bind* restricts a field to value r, while the bind-not operator restricts the field to any value but r. The *eq* restricts the result to rows whose fields $c_1, \ldots, c_n$ all have the same value. The rather complex projection operator is a combination of SQL's `SELECT` and `GROUP BY`.

A variable $x \in \mathit{Variables}$ represents a table in the underlying database. An identifier $c \in \mathit{Identifiers}$ names a column (or attribute, or field if you prefer) of a table. In practice, an identifier matches the regular expression

$$(a-z)(a-z|.)^*$$

where $a-z$ matches any lower case letter and dot matches only the period sign. Dots will be used to reflect subquery nesting. We shall often need to append a literal suffix beginning with a dot like ".from" or ".to" to a given identifier c. We write the extended identifiers as c_{from} or c_{to} respectively. **Warning:** *This notation should not be confused with indexed identifiers like c_i, c_k or c_n. An index of an identifier will always be a single letter, while suffixes in the above notation will always consist of several letters.*

Note that the sum expression accepts zero or more columns. For each row in the sum, the specified columns are multiplied. When there are zero columns, this product is defined to be 1, and we write the expression as *sum* 1. This special case is equivalent to the *count** expression in SQL.

Note that we use the word "table" about both tables in the underlying database (database tables) and tables as results of queries (views).

9.2 Parsing MORI/SQL to MORI algebra

As explained in the previous chapter, MORI/SQL is parsed into MORI algebra. This process is described formally by the denotational semantics in Figure 9.2.

There are a few things to notice in the figure. First of all, the sets in Roman font are syntactic categories of the MORI/SQL grammar. Second, we only represent the actual query part of a relation as a term in MORI algebra. The header of the relation and the `ORDER BY` declaration are silently passed on to the code generating part of the compiler. Third, the syntaxes for expressions in the two languages are almost identical, so we take the liberty of copying MORI/SQL expressions directly into the result term. The one exception is the `COUNT*` expression in MORI/SQL which – as mentioned above – can be represented by the $(\textit{sum}\ 1)$ expression in MORI algebra. Fourth, to be correct a MORI/SQL relation must `GROUP BY` the identifier that it `SELECT`s as the first.

One detail has been left out of the semantics: to keep our notation more compact in this chapter, we rename the rows of the *ILS* (see Section 8.1.1). In this chapter

- *from* is the row previously called *subjectID*,
- *to* is the row previously called *objectID*,
- *ts* is the row previously called *timestamp*, and
- *score* is the row previously called *size*.

Other than this, the semantics should be straightforward. It is worth noting that the only $\Pi_{\cdot/\cdot}$ operator in the resulting query is the outermost one. This will be useful when optimizing the query.

9.3 Denotational semantics

We shall formalize the meaning of terms in MORI algebra with a denotational semantics. First we'll need a common understanding of finite maps.

$[\![\cdot]\!]_{query}$	:	query $\rightarrow$ *Query*
$[\![$`SELECT` c `AS` c', L		
`FROM` F		
`WHERE` W		
`GROUP BY` c $]\!]_{query}$	$=$	$\Pi_{[\![L]\!]_{expl}/c'=c}([\![W]\!]_{wherel}\,[\![F]\!]_{froml})$
$[\![\cdot]\!]_{expl}$	:	explist $\rightarrow$ *Defs*
$[\![E_1$ `AS` $c_1, \ldots, E_n$ `AS` $c_n]\!]_{exp}$	$=$	$c_1 = E_1\ \cdots\ c_n = E_n$
		; *substituting* $(sum\ 1)$ *for* `COUNT`$*$
$[\![\cdot]\!]_{froml}$	:	fromlist $\rightarrow$ *Query*
$[\![x_1\,c_1 \cdots x_n\,c_n]\!]_{\mathcal{F}}$	$=$	$x_1(c_1) \times \cdots \times x_n(c_n)$
$[\![\cdot]\!]_{wherel}$	:	wherelist $\rightarrow$ *Query* $\rightarrow$ *Query*
$[\![con_1$ `AND` $\cdots$ `AND` $con_n]\!]_{wherel}\,Q$	$=$	$[\![con_1]\!]_{cstr} \circ \cdots \circ [\![con_n]\!]_{cstr}\,Q$
$[\![\cdot]\!]_{cstr}$	:	constraint $\rightarrow$ *Query* $\rightarrow$ *Query*
$[\![$`FOCUS` $=$ $c]\!]_{cstr}\,Q$	$=$	$\sigma_{bind(c)}\,Q$
$[\![c$ $=$ `FOCUS`$]\!]_{cstr}\,Q$	$=$	$\sigma_{bind(c)}\,Q$
$[\![$`FOCUS` $<>$ $c]\!]_{cstr}\,Q$	$=$	$\sigma_{bind\neg(c)}\,Q$
$[\![c$ $<>$ `FOCUS`$]\!]_{cstr}\,Q$	$=$	$\sigma_{bind\neg(c)}\,Q$
$[\![c = c']\!]_{cstr}\,Q$	$=$	$\sigma_{eq(cc')}\,Q$

Figure 9.2: Parsing a MORI/SQL query into a MORI algebra query.

Finite maps A *finite map* from set A to set B is a function

$$f : A \to B \cup \{undef\}$$

such that

$$dom f \overset{def}{=} f^{-1}(B)$$

is finite. We say that

$$f \in \text{FiniteMaps}(A,B)$$

For any choice of A and B, ε denotes the map for which $\varepsilon(a) = undef$ for all $a \in A$.

Whenever f is a finite map from A to B, $a_1, \ldots, a_n \in A$ and $b_1, \ldots, b_n \in B \cup \{undef\}$ we define

$$\begin{array}{lllll} f[a_1 \mapsto b_1, \cdots, a_n \mapsto b_n] = \lambda a. & \mathit{if}\ (a = a_n) & \mathit{then} & b_n & \mathit{else} \\ & \ldots & & & \\ & \mathit{if}\ (a = a_1) & \mathit{then} & b_1 & \mathit{else} \\ & f(a) & & & \end{array}$$

Note that the as do not have to be distinct. In case of multiple updates of the same a, the last one will get priority. Finally, if $f_1, \ldots, f_n \in \text{FiniteMaps}(A,B)$ all have disjoint domains we define

$$\begin{array}{lllll} f_1 \uplus \cdots \uplus f_n = \lambda a. & \mathit{if}\ (f_1(a) \in B) & \mathit{then} & f_1(a) & \mathit{else} \\ & \ldots & & & \\ & \mathit{if}\ (f_n(a) \in B) & \mathit{then} & f_n(a) & \mathit{else} \\ & \mathit{undef} & & & \end{array}$$

The union is *not* defined if the domains are not disjoint.

Schemas and Tables In a relational database table, all rows in a table must match the same schema, i.e. have the same number of columns with the same names and value types. A database schema can be seen as a type for all of its rows. The schema maps column identifiers to base types.

In our system, we'll only need three base types: $\mathcal{R}$, $\mathcal{T}$ and $\mathcal{N}$. When a column has base type $\mathcal{R}$, the values in that column come from a set *Resources*, which is a countably infinite index set. When a column has base type $\mathcal{T}$, the values in that column come from a set *TimeStamps* and indicate date and time of an event. When a column has base type $\mathcal{N}$, the values in that column are natural numbers. To indicate this correspondence, we define

$$\begin{aligned} \widehat{\mathcal{R}} &= \mathit{Resources} \\ \widehat{\mathcal{T}} &= \mathit{TimeStamps} \\ \widehat{\mathcal{N}} &= \mathbf{N} \end{aligned}$$

We model a schema as a finite map from the set of identifiers to our base types:

$$\tau \in \mathit{Schemas} = \mathit{FiniteMaps}(\mathit{Identifiers}, \{\mathcal{R}, \mathcal{T}, \mathcal{N}\})$$

The set of tables conforming to a specific schema τ is formally defined as follows:

$$\begin{aligned}\mathit{Tables}(\tau) = \{\{\rho_1, \ldots, \rho_n\} \mid\ & \forall i.\rho_i \in \mathit{FiniteMaps}(\ \mathit{Identifiers}, \\ & \qquad\qquad (\mathit{Resources} \cup \mathit{TimeStamps} \cup \mathbf{N})) \wedge \\ & \forall i.\mathit{dom}\ \rho_i = \mathit{dom}\ \tau \wedge \\ & \forall i \forall c \in \mathit{dom}\ \tau.\rho_i(c) \in \widehat{\tau(c)}\}\end{aligned}$$

It may seem weird that we don't define the ρs as mappings rather than *FiniteMap*s given the second and third conditions. The reason is that it will come in handy when we define our semantics: the more general set of ρs allows us to derive one table from another table of different schema more easily.

The set of *all* relational database tables is

$$\mathit{Tables} = \{T \mid \exists \tau \in \mathit{Schemas}.T \in \mathit{Tables}(\tau)\}$$

Definitions for the semantics The underlying database Θ is a mapping from variable names to relational database tables. In our (domain-specific) case, *all* tables in the underlying database have the same schema with four columns:

- *from* of type $\mathcal{R}$,
- *to*, also of type $\mathcal{R}$,
- *ts* of type $\mathcal{T}$, and
- *score* of type $\mathcal{N}$.

We'll call such a table an *event table* (because they're used to register events that have occurred). So

$$\begin{aligned}&\mathit{EventTables} = \mathit{Tables}(\varepsilon[\mathit{from} \mapsto \mathcal{R}, \mathit{to} \mapsto \mathcal{R}, \mathit{ts} \mapsto \mathcal{T}, \mathit{score} \mapsto \mathcal{N}]) \\ &\Theta : \mathit{Variables} \to \mathit{EventTables}\end{aligned}$$

The semantics of queries is given by a denotation of this form:

$$[\![\cdot]\!] : \mathit{Query} \to \mathit{Resources} \to (\mathit{Variables} \to \mathit{EventTables}) \to \mathit{Tables}$$

The semantics of expressions is given by a denotation

$$[\![\cdot]\!]_\varepsilon : \mathit{Expr} \to \mathit{Tables} \to \mathit{Resource} \cup \mathit{TimeStamps} \cup \mathbf{N}$$

Denotational semantics

$$
\begin{array}{lcl}
[\![x(c)]\!]r_0\Theta & = & \{\varepsilon[\ c_{from} \mapsto \rho(from), \\
& & \quad c_{to} \mapsto \rho(to), \\
& & \quad c_{ts} \mapsto \rho(ts), \\
& & \quad c_{score} \mapsto \rho(score)] \mid \rho \in \Theta(x)\} \\
[\![Q_1 \times \cdots \times Q_n]\!]r_0\Theta & = & \{\rho_1 \uplus \cdots \uplus \rho_n \mid \rho_i \in [\![Q_i]\!]r_0\Theta\} \\
[\![\sigma_{bind(c)}\, Q]\!]r_0\Theta & = & \{\rho \in [\![Q]\!]r_0\Theta \mid \rho(c) = r_0\} \\
[\![\sigma_{bind\neg(c)}\, Q]\!]r_0\Theta & = & \{\rho \in [\![Q]\!]r_0\Theta \mid \rho(c) \neq r_0\} \\
[\![\sigma_{eq(c_1,\ldots,c_n)}\, Q]\!]r_0\Theta & = & \{\rho \in [\![Q]\!]r_0\Theta \mid \rho(c_1) = \cdots = \rho(c_n)\} \\
[\![\Pi_{(c_1=E_1 \cdots c_n=E_n)/c'=c}\, Q]\!]r_0\Theta & = & \;\; let \quad T \;=\; [\![Q]\!]r_0\Theta \\
& & \;\; in\, \{\varepsilon[c' \mapsto \rho(c), \\
& & \qquad c_1 \mapsto [\![E_1]\!]_\varepsilon\{\rho' \in T \mid \rho'(c) = \rho(c)\}, \\
& & \qquad \cdots, \\
& & \qquad c_n \mapsto [\![E_n]\!]_\varepsilon\{\rho' \in T \mid \rho'(c) = \rho(c)\}] \mid \rho \in T\}
\end{array}
$$

The group-by operator is – as always – the most complex one to define. In brief, the subquery Q is evaluated. The rows of the resulting table T are then partitioned into equivalence classes; maximal subsets of T where all members have the same value r in the field c. Finally, the expressions are used to aggregate values of other fields, so that each equivalence class becomes one row in the final result. The renaming of c to c' is just a service to the programmer, inherited from MORI/SQL. The two identifiers do not have to be different.

A thing to notice is that the semantics of Cartesian product is only defined if the constituent queries produce tables with disjoint schemas. In other words: if two of the subqueries define the same identifier (attribute name), the product is not defined. This convention forces the query writer to be aware of name spaces. The type system below checks that this is indeed the case for well-typed queries. The type system will also check that all fields that are made subject to equalities or inequalities are of resource type (so we avoid considerations about equalities between elements of different value sets).

The semantics of expressions is defined by:

$$
\begin{array}{lcl}
[\![max\ c]\!]_\varepsilon S & = & \max_{\rho \in S} \rho(c) \\
[\![sum\ c_1 \cdots c_n]\!]_\varepsilon S & = & \sum_{\rho \in S} \rho(c_1) \cdots \rho(c_n) \\
[\![E_1 * E_2]\!]_\varepsilon S & = & ([\![E_1]\!]_\varepsilon S)([\![E_2]\!]_\varepsilon S) \\
[\![E_1 / E_2]\!]_\varepsilon S & = & ([\![E_1]\!]_\varepsilon S)/([\![E_2]\!]_\varepsilon S)
\end{array}
$$

where $max\ : \mathcal{P}(TimeStamps) \to TimeStamps$ operates on time stamps. For convenience we assume the existence of a special value $\infty_T \in TimeStamps$ for which $\max_{t \in \emptyset} t = \infty_T$. To avoid concerns regarding division by zero, we also assume the existence of $\infty_N \in \mathbf{N}$ and define $n/0 = \infty_N$ for all $n \in \mathbf{N}$.

Please note that, in the rule for sum expressions, when $n = 0$ (i.e. the list of cs to multiply is empty), we consider the product to be 1 for every row.

9.4 Types

The type τ of a query Q is a schema: a finite map from identifiers to base types (resource, time stamp or natural number):

$$\tau \in \text{FiniteMaps}(\mathit{Identifiers}, \{\mathcal{R}, \mathcal{T}, \mathcal{N}\})$$

This relation is simply written as

$$Q : \tau$$

Note that the type of a query is completely synthesized (constructed from Q's parts) and context-independent. It is not quite so for the type β of an expression E. This type is either $\mathcal{T}$ or $\mathcal{N}$, but only if applied in a schema for which E makes sense. We write this relation as

$$\tau \vdash E : \beta$$

Here's the type system for queries:

$$\frac{}{x(c) : \varepsilon[c_{from} \mapsto \mathcal{R}, c_{to} \mapsto \mathcal{R}, c_{ts} \mapsto \mathcal{T}, c_{score} \mapsto \mathcal{N}]}$$

$$\frac{Q_1 : \tau_1 \quad \cdots \quad Q_n : \tau_n}{Q_1 \times \cdots \times Q_n : \tau_1 \uplus \cdots \uplus \tau_n}$$

$$\frac{Q : \tau \quad \tau(c) = \mathcal{R}}{\sigma_{bind(c)}\, Q : \tau}$$

$$\frac{Q : \tau \quad \tau(c) = \mathcal{R}}{\sigma_{bind\neg(c)}\, Q : \tau}$$

$$\frac{Q : \tau \quad \tau(c_1) = \mathcal{R} \quad \cdots \quad \tau(c_n) = \mathcal{R}}{\sigma_{eq(c_1,\ldots,c_n)}\, Q : \tau}$$

$$\frac{i \neq j \Rightarrow c_i \neq c_j \quad Q : \tau \quad \tau(c) = \mathcal{R} \quad \forall i > 0. \tau \vdash E_i : \beta_i}{\Pi_{(c_1 = E_1 \cdots c_n = E_n)/c_0 = c}\, Q : \varepsilon[c_0 \mapsto \mathcal{R}, c_1 \mapsto \beta_1, \cdots, c_n \mapsto \beta_n]}$$

Note in particular that a Cartesian product can only be typed if its constituent tables (its factors) define disjoint sets of column identifiers. Note also the c and c_0

must be of resource type. Although the semantics can work even if they are not, this limitation is useful during optimization.

Here’s the type system for expressions:

$$\frac{\tau(c) = \mathcal{T}}{\tau \vdash \mathit{max}\ c : \mathcal{T}}$$

$$\frac{\forall i : \tau(c_i) = \mathcal{N}}{\tau \vdash \mathit{sum}\ c_1 \cdots c_n\ : \mathcal{N}}$$

$$\frac{\tau \vdash E_1 : \mathcal{N} \quad \tau \vdash E_2 : \mathcal{N}}{\tau \vdash E_1 * E_2 : \mathcal{N}} \qquad \frac{\tau \vdash E_1 : \mathcal{N} \quad \tau \vdash E_2 : \mathcal{N}}{\tau \vdash E_1 / E_2 : \mathcal{N}}$$

9.4.1 Query equivalence

We’re ready for defining semantical equivalence of queries, a concept we shall find useful later on. We begin by stating the following fact.

Observation 3 (Well-typed programs don’t go wrong). *Let $Q : \tau$. For any $r \in$ Resources and Θ : Variables $\to$ EventTables: $[\![Q]\!]r\Theta \in \mathit{Tables}(\tau)$.*

This makes our job easier as we are only interested in equivalence of well-typed queries.

Definition 8 (Equivalence). Let $Q_1 : \tau_1$ and $Q_2 : \tau_2$ be well-typed queries. We say that Q_1 and Q_2 are equivalent and write $Q_1 \equiv Q_2$ if and only if $\tau_1 = \tau_2$ and

$$\forall r \in \mathit{Resources}\ \forall\Theta :\ \mathit{Variables} \to \mathit{EventTables}. [\![Q_1]\!]r\Theta = [\![Q_2]\!]r\Theta$$

Relation $\equiv$ is clearly an equivalence (reflexive, symmetric and transitive).

9.5 Optimization

This section proves some equivalence results for queries. These results show how one can modify queries without modifying their semantics. We do not give an optimization algorithm as such.

Our optimizations are of two kinds. The first kind of optimization is to move bind- and bind-not-restrictions “downwards” in a query term. The point is that the earlier we perform these restrictions, the fewer rows we’ll have to deal with in the intermediate result tables. It is of special importance to try to move these restrictions below product operators, as these are the ones causing blowups in result table sizes.

The second kind of optimization is the introduction of extra GROUP-BYs. This is important as it also (drastically) reduces table sizes in practice. Doing this kind of optimization without changing the semantics of the query is quite subtle but often possible, as it will turn out.

Example The following sections will prove the following equivalences

$$\Pi_{s=(sum\ 1)/o=c'_{to}}\, \sigma_{bind(c_{from})}\, \sigma_{eq(c_{to}\, c'_{from})}\, (x(c) \times x'(c')) \qquad \equiv$$

$$\Pi_{s=(sum\ 1)/o=c'_{to}}\, \sigma_{eq(c_{to}\, c'_{from})}\, (x'(c') \times \sigma_{bind(c_{from})}\, x(c)\,) \qquad \equiv$$

$$\Pi_{s=(sum\ c_{cnt})/o=c'_{to}}\, \sigma_{eq(c_{to}\, c'_{from})} \Big(x'(c') \times \Pi_{c_{cnt}=(sum\ 1)/c_{to}=c_{to}}\, \sigma_{bind(c_{from})}\, x(c) \Big)$$

In what way is the latter query more efficient (optimized) than the first? The key point to efficiency is that the intermediate results, the tables that are computed for subexpressions of the query, have as few rows as possible. From the semantics of queries we get that

$$\begin{array}{rcl} |[\![Q_1 \times Q_2]\!] r_0 \Theta| & = & |[\![Q_1]\!] r_0 \Theta| \cdot |[\![Q_2]\!] r_0 \Theta| \\ \left|[\![\sigma_{bind(c)}\, Q]\!] r_0 \Theta\right| & \leq & |[\![Q]\!] r_0 \Theta| \\ \left|[\![\Pi_{D/c'=c}\, Q]\!] r_0 \Theta\right| & \leq & |[\![Q]\!] r_0 \Theta| \end{array}$$

Assuming that all tables in the underlying database Θ are very large, the above inequalities will most often mean difference by orders of magnitude. Applying this to the example, we realize that the second query should be more efficient than the first, and that the third should be even more efficient.

Section 9.5.1 proves the first equivalence above. Section 9.5.2 proves the second equivalence.

9.5.1 Restriction propagation

Definition 9. For notational convenience we introduce two new restrictions:

$$\begin{array}{rcl} \sigma_{triv(c)}\, Q & = & Q \\ \sigma_{fail(c)}\, Q & = & \sigma_{bind(c)}\, \sigma_{bind\neg(c)}\, Q \end{array}$$

where c is an identifier. These restrictions are only syntactic sugar – the left-hand sides are lexical alternatives to the corresponding right-hand sides. There is thus no need to define types or semantics for these constructs.

$\sigma_{P_1}\sigma_{P_2}Q \equiv ?$	$P_2 = bind(c)$	$P_2 = bind\neg(c)$	$P_2 = triv(c)$	$P_2 = fail(c)$
$P_1 = bind(c)$	$\sigma_{bind(c)}\,Q$	$\sigma_{fail(c)}Q$	$\sigma_{bind(c)}\,Q$	$\sigma_{fail(c)}Q$
$P_1 = bind\neg(c)$	$\sigma_{fail(c)}Q$	$\sigma_{bind\neg(c)}\,Q$	$\sigma_{bind\neg(c)}\,Q$	$\sigma_{fail(c)}Q$
$P_1 = triv(c)$	$\sigma_{bind(c)}\,Q$	$\sigma_{bind\neg(c)}\,Q$	$\sigma_{triv(c)}Q$	$\sigma_{fail(c)}Q$
$P_1 = fail(c)$	$\sigma_{fail(c)}Q$	$\sigma_{fail(c)}Q$	$\sigma_{fail(c)}Q$	$\sigma_{fail(c)}Q$

Figure 9.3: Algebraic laws for restrictions. We assume $Q : \tau$ and $\tau(c) = \mathcal{R}$. The query $\sigma_{P_1}\sigma_{P_2}Q$ is equivalent to the query in the corresponding table entry.

Note that when $Q : \tau$ and $\tau(c) = \mathcal{R}$ we have the laws shown in Figure 9.3.

Recall that any query produced by the MORI/SQL parser (Section 9.2) contains only one $\Pi_{\cdot/\cdot}$ operator, and that this operator is the outermost one. This observation will allow to focus on rewriting queries without the $\Pi_{\cdot/\cdot}$ operator.

Definition 10. A query Q is in *bind-normal-form* if and only if Q is well-typed and

$$Q = \sigma_{eq(c_1' c_1'')} \cdots \sigma_{eq(c_k' c_k'')} \left(\begin{array}{lc} \sigma_{P_{11}((c_1)_{from})}\sigma_{P_{12}((c_1)_{to})}x_1(c_1) & \times \\ \cdots & \times \\ \sigma_{P_{n1}((c_n)_{from})}\sigma_{P_{n2}((c_n)_{to})}x_n(c_n) & \end{array} \right)$$

where $\forall i,j : P_{ij} \in \{\sigma_{bind()}, \sigma_{bind\neg()}, \sigma_{triv}(), \sigma_{fail}()\}$.

Theorem 4. *If Q is well-typed and does not contain any $\Pi_{\cdot/\cdot}$-terms, there exists Q' such that $Q \equiv Q'$ and the latter is in bind-normal-form.*

Proof. The proof is performed by structural induction on Q and uses a number of algebraic laws that will be given below.

Case $Q = x(c)$: This case is simple as $x(c) \equiv \sigma_{triv}(c_{from})\sigma_{triv}(c_{to})x(c)$.

Case $Q = Q_1 \times \cdots \times Q_n$: Assume by induction that $Q_1', \ldots, Q_n'$ are in bind-normal-form and $Q_1 \equiv Q_1', \ldots, Q_n \equiv Q_n'$. The theorem holds by the distributivity of restrictions (Lemma 4) and associativity of Cartesian product (Lemma 6).

Case $Q = \sigma_{bind(c)}\,Q'$ or $Q = \sigma_{bind\neg(c)}\,Q'$: Assume by induction that $Q' \equiv Q''$ and Q'' is in bind-normal-form. The theorem holds by the commutativity (Lemma 5) and distributivity (Lemma 4) of restrictions.

Case $Q = \sigma_{eq(c_1 \cdots c_n)}\, Q'$: Assume by induction that $Q' \equiv Q''$ and Q'' is in bind-normal-form. The theorem holds as $\sigma_{eq(c_1 \cdots c_n)}\, Q''$ is obviously equivalent to

$$\sigma_{eq(c_1 c_2)} \cdots \sigma_{eq(c_{n-1} c_n)}\, Q''$$

□

Corollary 2. *If Q is well-typed and $Q = \Pi_{D/c'=c}\, Q'$ where Q' does not contain any $\Pi_{\cdot/\cdot}$-terms, there exists Q'' such that $Q \equiv \Pi_{D/c'=c}\, Q''$ and Q'' is in bind-normal-form.*

Lemma 4 (Distributivity of restrictions). *When $Q \times Q'$ is well-typed, $Q' : \tau'$, and $\tau'(c') = \mathcal{R}$ then*

$$\begin{array}{rcl} \sigma_{bind(c')}\,(Q \times Q') & \equiv & Q \times \sigma_{bind(c')}\, Q' \\ \sigma_{bind\neg(c')}\,(Q \times Q') & \equiv & Q \times \sigma_{bind\neg(c')}\, Q' \end{array}$$

Proof. Let Θ and r_0 be given. Depending on whether the restriction is $\sigma_{bind()}$ or $\sigma_{bind\neg()}$ we define

$$\begin{array}{rcl} P(\rho) & \Leftrightarrow & \rho(c') = r_0 \\ & \text{or} & \\ P(\rho) & \Leftrightarrow & \rho(c') \neq r_0 \end{array}$$

The semantics of the left-hand side query is given by

$$\{\rho \uplus \rho' \mid \rho \in [\![Q]\!] r_0 \Theta, \rho' \in [\![Q']\!] r_0 \Theta, P(\rho \uplus \rho')\}$$

which, because $\tau(c')$ must be *undef*, is equal to

$$\{\rho \uplus \rho' \mid \rho \in [\![Q]\!] r_0 \Theta, \rho' \in [\![Q']\!] r_0 \Theta, P(\rho')\}$$

i.e. the semantics of the right-hand side query. □

Lemma 5 (Commutativity of restrictions). *Let $Q : \tau$ and $\tau(c) = \tau(c') = \tau(c_1) = \cdots = \tau(c_n) = \mathcal{R}$. The following equivalences hold:*

$$\begin{array}{rcl} \sigma_{bind(c)}\,\sigma_{bind\neg(c')}\, Q & \equiv & \sigma_{bind\neg(c')}\,\sigma_{bind(c)}\, Q \\ \sigma_{bind(c)}\,\sigma_{eq(c_1 \cdots c_n)}\, Q & \equiv & \sigma_{eq(c_1 \cdots c_n)}\,\sigma_{bind(c)}\, Q \\ \sigma_{bind\neg(c)}\,\sigma_{eq(c_1 \cdots c_n)}\, Q & \equiv & \sigma_{eq(c_1 \cdots c_n)}\,\sigma_{bind\neg(c)}\, Q \end{array}$$

Proof. Trivial. □

Lemma 6 (Associativity and commutativity of Cartesian product). *When*

$$Q_1 \times \cdots \times Q_n \times Q'_1 \times \cdots \times Q'_m$$

is well-typed we have

$$\begin{aligned} Q_1 \times \cdots \times Q_n \times Q'_1 \times \cdots \times Q'_m &\equiv (Q_1 \times \cdots \times Q_n) \times Q'_1 \times \cdots \times Q'_m \\ Q_1 \times \cdots \times Q_n \times Q'_1 \times \cdots \times Q'_m &\equiv Q'_1 \times \cdots \times Q'_m \times Q_1 \times \cdots \times Q_n \end{aligned}$$

Proof. Trivial. □

9.5.2 GROUP-BY introduction

We now move on to the subject of when and how $\Pi_{\cdot/\cdot}$ operators may be inserted into a query $Q_0 = \Pi_{D/c'=c} Q$, where Q is in bind-normal-form, without changing the semantics of Q_0. The approach we describe in this section is closely related to the *Generalized Coalescing Grouping* principle outlined by Chaudhuri and Shim in [CS94]. The following is an important concept in our investigation:

Definition 11 (Observational equivalence). Let $Q : \tau$ and $Q' : \tau'$ be well-typed queries and E and E' be expressions such that $\tau \vdash E : \beta$ and $\tau' \vdash E' : \beta$. When $M = \{c_1, \ldots, c_n\}$ is a set of identifiers for which $M \subseteq \tau^{-1}(\mathcal{R}) \cup \tau'^{-1}(\mathcal{R})$, we say that (E, Q) and (E', Q') are observationally equivalent on M,

$$(E, Q) \sim_M (E', Q')$$

if and only if

$$\begin{aligned} \forall\Theta\,\forall r_0, r_1, \ldots, r_n : \quad & [\![E]\!]_\varepsilon \{\rho \in [\![Q]\!] r_0 \Theta \mid \rho(c_1) = r_1 \wedge \cdots \wedge \rho(c_n) = r_n\} = \\ & [\![E']\!]_\varepsilon \{\rho' \in [\![Q']\!] r_0 \Theta \mid \rho(c_1) = r_1 \wedge \cdots \wedge \rho(c_n) = r_n\} \end{aligned}$$

Notice that $\sim$ *is* an equivalence: it is reflexive, symmetric and transitive. It is weaker than $\equiv$ in the sense that $Q : \tau$ and $Q \equiv Q'$ implies $(E, Q) \sim_{\tau^{-1}(\mathcal{R})} (E, Q')$.

Definition 12. For convenience and clarity we introduce the following notation:

$$\mathit{isEmpty}? = (\mathit{sum}\ 1)/(\mathit{sum}\ 1)$$

That rather peculiar expression has a nice property that we shall need:

$$[\![\mathit{isEmpty}?]\!]_\varepsilon S = 1 \Leftrightarrow S \neq \emptyset$$

(It is ∞_N when the set S is empty.)

The following theorem can be used when we want to rewrite $\Pi_{D/c'=c} Q$ into $\Pi_{D/c'=c} Q'$, specifically when Q is in bind-normal-form and Q' is similar to Q but has a $\Pi_{\cdot/\cdot}$ operator in one of its subterms.

Theorem 5. *If* $(\mathit{isEmpty?}, Q) \sim_{\{c\}} (\mathit{isEmpty?}, Q')$ *and* $\forall i : (E_i, Q) \sim_{\{c\}} (E_i', Q')$ *then*

$$\Pi_{c_1=E_1 \cdots c_n=E_n/c'=c}\, Q \equiv \Pi_{c_1=E_1' \cdots c_n=E_n'/c'=c}\, Q'$$

Proof. It is clear that the two queries will be well-typed and have the same type. Let Θ and r_0 be given and let

$$\begin{aligned} T_Q &= [\![Q]\!] r_0 \Theta \\ T_{Q'} &= [\![Q']\!] r_0 \Theta \\ T_{\mathit{left}} &= [\![\Pi_{c_1=E_1 \cdots c_n=E_n/c'=c}\, Q]\!] r_0 \Theta \\ T_{\mathit{right}} &= [\![\Pi_{c_1=E_1' \cdots c_n=E_n'/c'=c}\, Q']\!] r_0 \Theta \end{aligned}$$

Assume for the sake of contradiction that $T_{\mathit{left}} \neq T_{\mathit{right}}$. This implies the existence of a ρ such that $\rho \in T_{\mathit{left}}$ but $\rho \notin T_{\mathit{right}}$ (or the other way around, but this situation is completely symmetrical). Let $r = \rho(c)$. From the semantics of queries and the assumptions we get that

$$\begin{aligned} \rho &= \varepsilon[c' \mapsto r, \\ &\quad c_1 \mapsto [\![E_1]\!]_\varepsilon \{\rho' \in T_Q \mid \rho'(c) = r\}, \\ &\quad \ldots, \\ &\quad c_n \mapsto [\![E_n]\!]_\varepsilon \{\rho' \in T_Q \mid \rho'(c) = r\}] \\ &= \varepsilon[c' \mapsto r, \\ &\quad c_1 \mapsto [\![E_1']\!]_\varepsilon \{\rho' \in T_{Q'} \mid \rho'(c) = r\}, \\ &\quad \ldots, \\ &\quad c_n \mapsto [\![E_n']\!]_\varepsilon \{\rho' \in T_{Q'} \mid \rho'(c) = r\}] \end{aligned}$$

By the semantics, the latter is in T_{right} if and only if there is a ρ' in $T_{Q'}$ such that $\rho'(c) = r$. Such a ρ' does exists, as

$$1 = [\![\mathit{isEmpty?}]\!]_\varepsilon \{\rho'' \in T_Q \mid \rho''(c) = r\} = [\![\mathit{isEmpty?}]\!]_\varepsilon \{\rho'' \in T_{Q'} \mid \rho''(c) = r\}$$

Thus, $\rho \in T_{\mathit{right}}$ and we have a contradiction. □

Lemma 7. *If*

$$\begin{gathered} (E_1, Q) \sim_M (E_1', Q') \wedge \\ (E_2, Q) \sim_M (E_2', Q') \end{gathered}$$

then

$$\begin{gathered} ((E_1 * E_2), Q) \sim_M ((E_1' * E_2'), Q') \wedge \\ ((E_1 / E_2), Q) \sim_M ((E_1' / E_2'), Q') \end{gathered}$$

Proof. Trivial. □

Lemma 7 has as consequence that we only need to establish observational equivalences for *ground expressions*:

Definition 13. A expression E is *ground* if $E = max\ c$ or $E = sum\ c_1 \cdots c_n$. In the rest of this section, all expressions are expected to be ground.

If the query Q that we wish to optimize is on bind-normal-form, its outermost subterms are field equality restrictions. Therefore we need the following two theorems:

Theorem 6. *If* $(E,Q) \sim_{\{c,c'\}} (E',Q')$ *then*

$$(E, \sigma_{eq(cc')}\, Q) \sim_{\{c,c'\}} (E', \sigma_{eq(cc')}\, Q')$$

when E *and* E' *are ground.*

Proof. Let Θ, r_0, r and r' be given. If $r = r'$ then

$$\begin{array}{ll} [\![E]\!]_\varepsilon\{\rho \in [\![\sigma_{eq(cc')}\, Q]\!] r_0 \Theta \mid \rho(c) = r \wedge \rho(c') = r'\} & = \\ [\![E]\!]_\varepsilon\{\rho \in [\![Q]\!] r_0 \Theta \mid \rho(c) = r \wedge \rho(c') = r\} & = \\ [\![E']\!]_\varepsilon\{\rho' \in [\![Q']\!] r_0 \Theta \mid \rho'(c) = r \wedge \rho'(c') = r\} & = \\ [\![E']\!]_\varepsilon\{\rho' \in [\![\sigma_{eq(cc')}\, Q']\!] r_0 \Theta \mid \rho'(c) = r \wedge \rho'(c') = r'\} & \end{array}$$

If $r \neq r'$ then

$$\begin{array}{ll} [\![E]\!]_\varepsilon\{\rho \in [\![\sigma_{eq(cc')}\, Q]\!] r_0 \Theta \mid \rho(c) = r \wedge \rho(c') = r'\} & = \\ [\![E]\!]_\varepsilon \emptyset & = \\ [\![E']\!]_\varepsilon \emptyset & = \\ [\![E]\!]_\varepsilon\{\rho' \in [\![\sigma_{eq(cc')}\, Q']\!] r_0 \Theta \mid \rho'(c) = r \wedge \rho'(c') = r'\} & \end{array}$$

The equality $[\![E]\!]_\varepsilon \emptyset = [\![E']\!]_\varepsilon \emptyset$ always holds when E and E' are ground and of the same type. □

Theorem 7. *If* $(E,Q) \sim_{M \cup \{c\}} (E',Q')$ *then* $(E,Q) \sim_M (E',Q')$ *when* E *and* E' *are ground.*

Proof. The types must clearly match. Let $M = \{c_1, \ldots, c_n\}$, and let $\Theta, r_0, r_1, \ldots, r_n$ be given.

$$\begin{array}{ll} [\![E]\!]_\varepsilon\{\rho \in [\![Q]\!] r_0 \Theta \mid \rho(c_1) = r_1 \wedge \cdots \wedge \rho(c_n) = r_n\} & = \\ [\![E]\!]_\varepsilon \bigcup_{r \in Resources} \{\rho \in [\![Q]\!] r_0 \Theta \mid \rho(\rho(c) = r \wedge c_1) = r_1 \wedge \cdots \wedge \rho(c_n) = r_n\} & = \\ [\![E']\!]_\varepsilon \bigcup_{r \in Resources} \{\rho' \in [\![Q']\!] r_0 \Theta \mid \rho'(c) = r \wedge \rho'(c_1) = r_1 \wedge \cdots \wedge \rho'(c_n) = r_n\} & = \\ [\![E']\!]_\varepsilon\{\rho' \in [\![Q]\!] r_0 \Theta \mid \rho'(c_1) = r_1 \wedge \cdots \wedge \rho'(c_n) = r_n\} & \end{array}$$

The middle equality holds because E and E' are ground and of the same type. □

We are now ready to show when and how GROUP BY can actually be performed on a query that is part of a Cartesian product. The next four theorems relate to this question, each one focusing on preserving observational equivalence for a different kind of expression.

Theorem 8. *Let $Q : \tau$ and $Q' : \tau'$ be queries such that $dom\,\tau \cap dom\,\tau' = \emptyset$, $\tau(c) = \mathcal{T}$ and $\tau'(c') = \mathcal{R}$. Then*

$$\big(max\;c\,, Q \times Q'\big) \sim_{\tau^{-1}(\mathcal{R}) \cup \{c'\}} \big(max\;c\,, Q \times \Pi_{\varepsilon/c'=c'}\,Q'\big)$$

Proof. It is clear that that both expressions are properly typable and will have the same type. Let Θ and $r_0, r_1, \ldots, r_{|\tau^{-1}(\mathcal{R})|}, r'$ be given. We define predicates P and P' on the set of all rows by

$$\begin{array}{rcl} P(\rho) & \Leftrightarrow & \rho(c_1) = r_1 \wedge \cdots \wedge \rho(c_{|\tau^{-1}(\mathcal{R})|}) = c_{|\tau^{-1}(\mathcal{R})|} \\ P'(\rho) & \Leftrightarrow & \rho(c') = r' \end{array}$$

where $c_1, \ldots, c_{|\tau^{-1}(\mathcal{R})|}$ is an enumeration of the elements in $\tau^{-1}(\mathcal{R})$.

We also define

$$\begin{array}{rcl} T_Q & = & [\![Q]\!] r_0 \Theta \\ T_{Q'} & = & [\![Q']\!] r_0 \Theta \\ T_{left} & = & \{\rho \uplus \rho' \mid \rho \in T_Q, \rho' \in T_{Q'}, P(\rho \uplus \rho') \wedge P'(\rho \uplus \rho')\} \\ T_\Pi & = & \{\varepsilon[c' \mapsto \rho'(c')] \mid \rho' \in T_{Q'}\} \\ T_{right} & = & \{\rho \uplus \rho' \mid \rho \in T_Q, \rho' \in T_\Pi, P(\rho \uplus \rho') \wedge P'(\rho \uplus \rho')\} \end{array}$$

A crucial fact is that there is a $\rho' \in T_{Q'}$ for which $P'(\rho')$ if and only if there is a $\rho' \in T_\Pi$ such $P'(\rho')$. We have

$$\begin{array}{rcl} \max\limits_{\rho'' \in T_{left}} \rho''(c) & = & \max\limits_{\substack{\rho \in T_Q \\ \rho' \in T_{Q'} \\ P(\rho \uplus \rho') \\ P'(\rho \uplus \rho')}} (\rho \uplus \rho')(c) \\ & = & \max\limits_{\substack{\rho \in T_Q \\ \rho' \in T_{Q'} \\ P(\rho) \\ P'(\rho')}} \rho(c) \\ & = & \max\limits_{\substack{\rho \in T_Q \\ \rho' \in T_\Pi \\ P(\rho) \\ P'(\rho')}} \rho(c) \\ & = & \max\limits_{\substack{\rho \in T_Q \\ \rho' \in T_\Pi \\ P(\rho \uplus \rho') \\ P'(\rho \uplus \rho')}} (\rho \uplus \rho')(c) \\ & = & \max\limits_{\rho'' \in T_{right}} \rho''(c) \end{array}$$

which proves the theorem. □

Theorem 9. *Let $Q : \tau$ and $Q' : \tau'$ be queries such that $dom\,\tau \cap dom\,\tau' = \emptyset$, $\tau'(c') = \mathcal{R}$ and $\tau'(c'') = \mathcal{T}$. Then*

$$\left(max\ c'', Q \times Q'\right) \sim_{\tau^{-1}(\mathcal{R})\cup\{c'\}} \left(max\ c'', Q \times \Pi_{c''=max\ c''/c'=c'}\, Q'\right)$$

Proof. The proof of this theorem resembles the proof of Theorem 8. Again it is clear that both expressions are properly typable and will have the same type. Let Θ and $r_0, r_1, \ldots, r_{|\tau^{-1}(\mathcal{R})|}, r'$ be given. We define $P, P', T_Q, T_{Q'}$ and T_{left} as in the proof of Theorem 8, but let

$$\begin{aligned} T_\Pi &= \{\varepsilon[c' \mapsto \rho'(c'), c'' \mapsto \max_{\substack{\rho'' \in T_{Q'} \\ \rho''(c') = \rho'(c')}} \rho''(c'')] \mid \rho' \in T_{Q'}\} \\ T_{right} &= \{\rho \uplus \rho' \mid \rho \in T_Q, \rho' \in T_\Pi, P(\rho \uplus \rho') \wedge P'(\rho \uplus \rho')\} \end{aligned}$$

In parallel to the above proof we now have

$$\begin{aligned} \max_{\rho'' \in T_{left}} \rho''(c'') &= \max_{\substack{\rho \in T_Q \\ \rho' \in T_{Q'} \\ P(\rho \uplus \rho') \\ P'(\rho \uplus \rho')}} (\rho \uplus \rho')(c'') \\ &= \max_{\substack{\rho \in T_Q \\ \rho' \in T_{Q'} \\ P(\rho) \\ P'(\rho')}} \rho'(c'') \\ &= \max_{\substack{\rho \in T_Q \\ \rho' \in T_\Pi \\ P(\rho) \\ P'(\rho')}} \rho'(c'') \\ &= \max_{\substack{\rho \in T_Q \\ \rho' \in T_\Pi \\ P(\rho \uplus \rho') \\ P'(\rho \uplus \rho')}} (\rho \uplus \rho')(c'') \\ &= \max_{\rho'' \in T_{right}} \rho''(c'') \end{aligned}$$

$\square$

Theorem 10. *Let $Q : \tau$ and $Q' : \tau'$ be queries such that $dom\,\tau \cap dom\,\tau' = \emptyset$, $\tau'(c') = \mathcal{R}$, $\tau(c'') = \tau'(c'') = undef$, and $\tau'(c'_1) = \cdots = \tau'(c'_n) = \mathcal{N}$ where $n > 0$. Then*

$$\left(sum\ c'_1 \cdots c'_n, Q \times Q'\right) \sim_{\tau^{-1}(\mathcal{R})\cup\{c'\}} \left(sum\ c'', Q \times \Pi_{c''=sum\ c'_1 \cdots c'_n/c'=c'}\, Q'\right)$$

Proof. This proof also resembles the proof of Theorem 8. Once again it is clear that both expressions are properly typable and will have the same type. Let Θ and $r_0, r_1, \ldots, r_{|\tau^{-1}(\mathcal{R})|}, r'$ be given. We define $P, P', T_Q, T_{Q'}$ and T_{left} as in the proof of Theorem 8, but this time we let

$$\begin{aligned} T_\Pi &= \{\varepsilon[c' \mapsto \rho'(c'), c'' \mapsto \sum_{\substack{\rho'' \in T_{Q'} \\ \rho''(c') = \rho'(c')}} \rho''(c'_1) \cdots \rho''(c'_n)] \mid \rho' \in T_{Q'}\} \\ T_{right} &= \{\rho \uplus \rho' \mid \rho \in T_Q, \rho' \in T_\Pi, P(\rho \uplus \rho') \wedge P'(\rho \uplus \rho')\} \end{aligned}$$

We derive

$$\begin{aligned} \sum_{\rho'' \in T_{left}} \rho''(c'_1) \cdots \rho''(c'_n) &= \sum_{\substack{\rho \in T_Q \\ \rho' \in T_{Q'} \\ P(\rho \uplus \rho') \\ P'(\rho \uplus \rho')}} (\rho \uplus \rho')(c'_1) \cdots (\rho \uplus \rho')(c'_n) \\ &= \sum_{\substack{\rho \in T_Q \\ \rho' \in T_{Q'} \\ P(\rho) \\ P'(\rho')}} \rho'(c'_1) \cdots \rho'(c'_n) \\ &= \sum_{\substack{\rho \in T_Q \\ \rho' \in T_\Pi \\ P(\rho) \\ P'(\rho')}} \rho'(c'') \\ &= \sum_{\substack{\rho \in T_Q \\ \rho' \in T_\Pi \\ P(\rho \uplus \rho') \\ P'(\rho \uplus \rho')}} (\rho \uplus \rho')(c'') \\ &= \sum_{\rho'' \in T_{right}} \rho''(c'') \end{aligned}$$

This proves the theorem. □

Theorem 11. *Let $Q : \tau$ and $Q' : \tau'$ be queries such that $dom\,\tau \cap dom\,\tau' = \emptyset$, $\tau'(c') = \mathcal{R}$, $\tau(c'') = \tau'(c'') = undef$, and $\tau(c_1) = \cdots = \tau(c_n) = \mathcal{N}$ where $n \geq 0$. Then*

$$\big(sum\ c_1 \cdots c_n, Q \times Q'\big) \sim_{\tau^{-1}(\mathcal{R}) \cup \{c'\}} \big(sum\ c''\, c_1 \cdots c_n, Q \times \Pi_{c''=sum\ 1/c'=c'}\, Q'\big)$$

Proof. This proof follows the same pattern as the above ones. It is clear that both expressions are properly typable and will have the same type. Let Θ and $r_0, r_1, \ldots, r_{|\tau^{-1}(\mathcal{R})|}, r'$ be given. We define $P, P', T_Q, T_{Q'}$ and T_{left} as in the proof of

Theorem 8, and let

$$\begin{aligned}
T_\Pi &= \{\varepsilon[c' \mapsto \rho'(c'), c'' \mapsto \sum_{\substack{\rho'' \in T_{Q'} \\ \rho''(c') = \rho'(c')}} 1] \mid \rho' \in T_{Q'}\} \\
T_{right} &= \{\rho \uplus \rho' \mid \rho \in T_Q, \rho' \in T_\Pi, P(\rho \uplus \rho') \wedge P'(\rho \uplus \rho')\}
\end{aligned}$$

This time the our derivation is as follows:

$$\begin{aligned}
\sum_{\rho'' \in T_{left}} \rho''(c_1)\cdots\rho''(c_n) &= \sum_{\substack{\rho \in T_Q \\ \rho' \in T_{Q'} \\ P(\rho \uplus \rho') \\ P'(\rho \uplus \rho')}} (\rho \uplus \rho')(c_1)\cdots(\rho \uplus \rho')(c_n) \\
&= \sum_{\substack{\rho \in T_Q \\ \rho' \in T_{Q'} \\ P(\rho) \\ P'(\rho')}} \rho(c_1)\cdots\rho(c_n) \\
&= \left(\sum_{\substack{\rho \in T_Q \\ P(\rho)}} \rho(c_1)\cdots\rho(c_n) \right) \left(\sum_{\substack{\rho' \in T_{Q'} \\ P'(\rho')}} 1 \right) \\
&= \left(\sum_{\substack{\rho \in T_Q \\ P(\rho)}} \rho(c_1)\cdots\rho(c_n) \right) \left(\sum_{\substack{\rho' \in T_\Pi \\ P'(\rho')}} \rho'(c'') \right) \\
&= \sum_{\substack{\rho \in T_Q \\ \rho' \in T_\Pi \\ P(\rho) \\ P'(\rho')}} \rho'(c'')\rho(c_1)\cdots\rho(c_n) \\
&= \sum_{\substack{\rho \in T_Q \\ \rho' \in T_\Pi \\ P(\rho \uplus \rho') \\ P'(\rho \uplus \rho')}} (\rho \uplus \rho')(c'')(\rho \uplus \rho')(c_1)\cdots(\rho \uplus \rho')(c_n) \\
&= \sum_{\rho'' \in T_{right}} \rho''(c'')\rho''(c_1)\cdots\rho''(c_n)
\end{aligned}$$

This proves the theorem. □

Note that in the above Theorem, the number *n* of factors may be 0. Together with Lemma 7 this case gives us an observational equivalence for the *isEmpty*? expression.

Thus we have shown how a GROUP BY can be introduced into $Q \times Q'$ in many situations. In fact, the one kind of expression we *cannot* handle is a sum of fields from *both* Q and Q'. The assignment in the GROUP BY that we add is not the same in all cases, but we can obviously unify the $\Pi_{\cdot/\cdot}$ operators simply by appending their assignment lists (assuming the assigned identifiers are different).

9.6 Conclusion

This chapter desribed the MORI algebra, a domain-specific algebra invented to serve as intermediate language in the compiler we described in Chapter 8.

Working with an abstract algebra proved itself extremely useful, not only as a stage between MORI/SQL and PL/SQL but especially for reasoning about optimizations. The theory in Sections 9.5.1 and 9.5.2 would have been awkward to develop and near impossible to implement on the SQL level.

The optimization we described in Section 9.5.2 is closely related to the *Generalized Coalescing Grouping* principle outlined by Chaudhuri and Shim in [CS94]. We generalized their results and proved correctness in a formal setting. Moreover, we proved our results for concrete aggregation functions.

We have a running implementation of an optimizer based on the principles we have described, and our use of it have confirmed the usefulness of the work.

Conclusion

10 - Conclusion

Give a man a fish and you feed him for a day.
Teach him how to fish and you feed him for a lifetime.
Lao Tzu
—
If you give a man a fish he will eat for a day.
But if you teach a man to fish he will buy an ugly hat.
And if you talk about fish to a starving man
then you're a consultant
Dogbert

In this chapter I will try to sum up the academic contributions of this thesis and the most interesting directions for future work.

10.1 Contributions

The thesis was divided into three separate parts. I believe that each part contains contributions to their own separate academic fields.

Part I was called *Domain-specific language theory*. It contained two surveys, each one of them relating to a particular aspect of domain-specific languages and their role in the software development process, and each of them associated with a discussion that compared methods of the reviewed papers. As a practitioner – working on concrete applications of domain-specific languages – I needed to get an overview of the literature on these two aspects. Both surveys came into existence because I could not find any papers that covered the same particular aspects of domain-specific languages (DSLs) in sufficient detail.

Chapter 2 reviewed many papers on methods for implementing DSLs. While indeed a lot of papers have been written on this topic, most of these papers seem to only mention one or very few methods, in contrast to my general survey. Chapter 3 covers a topic that seems to receive very little attention in the programming langauge community: how to compare the advantages and disadvantages of a given, new DSL to pre-existing alternatives. Some papers do sketch such evaluations, but the evaluation method employed differs considerably, and there is often no clear rationale for the choice of method in a given paper. I believe that Chapter 3 provides a first step towards comparing the different evaluation methods and reasoning about which method to choose in a given project.

Part II of the thesis, *Partial evaluation theory,* also made several contributions. Chapter 4 presented a new semantics for a subset of the language *Erlang*;

a semantics which models at least two important issues in *Erlang* that I have not seen formalized hitherto: explicit name-spaces for process IDs and ordered, asynchronous message transition. On top of this, the new semantics is also unusually modular and should thus be relatively easy to extend to larger subsets of the language.

Chapter 5 gave a number of motivating examples and potential benchmarks for partial evaluation of *Erlang*. The examples were also used as background for a formulation of *what is* specialization in perspective of the semantics from Chapter 4. The chapter also contained a survey of papers on partial evaluation of concurrent languages.

The last chapter in the second part of the thesis contributes to the theoretical understanding of differences and similarities between online and offline partial evaluation. Robert Glück and I proved that – in contrast to a common belief – for many languages the two approaches achieve the same accuracy in finding static information. The chapter has, in form of a regular paper, been accepted for publication by the *ACM Transactions on Programming Languages and Systems*.

Part III of this thesis was called *MORI: an application of domain-specific languages* and described my major example of how domain-specific languages can be used in practical software development. The three chapters thus add flesh and blood to the more abstract considerations of Part I, motivating the need for Chapters 2 and 3. Moreover, Chapter 9 demonstrates how very complicated optimizations of a domain-specific language can be developed and explained through formal semantics. And in the process, certain results pertaining to the correctness of relational database query optimizations are established by formal proof. To my knowledge, this is the most detailed formal treatment of those optimizations.

10.2 Future work

Evaluation of DSLs It seems clear that more work could be done in this area. Unfortunately, evaluations in the present literature more often resemble sales talks than whole-hearted attempts at objectively critizing the DSL approach in a given context. To gain a better understanding of the DSL approach to software development, we need more DSL papers to report on reasonably objective evaluations in details. We also need to compare surveys like Chapter 3 with other texts on evaluation in the software engineering literature.

Erlang semantics The semantics of Chapter 4 leaves room for improvement. Most obviously it should be extended to model more of the interesting features of *Erlang*, particularly modules and hot code replacement which should be of interest to the program transformation community. Modelling *Erlang*'s clever support for writing fail-safe servers would also be interesting, because formal inspections of fail-over handling could turn out to be quite valuable.

On a more theoretical line, I would like to see a version of the semantics based on observations of communication as in e.g. the π-calculus. I conjecture that this would simplify the definition of correct program transformations, as it would allow standard definitons of simulation and bisimulation.

Partial evaluation of Erlang The future work on this topic is clear: write an actual partial evaluator for *Erlang*. There are still design issues to be solved, e.g. should the system be online of offline, are partially static data structures a *must*, and how do we practically handle intended nontermination? An important question is whether some of the related work that was reviewed in Chapter 5 can be applied in an *Erlang* setting. Due to the essential differences between concurrency paradigms, the answer to this question is not clear.

Polyvariant binding-time analyses What happens to the accuracy and termination properties of a polyvariant BTA, when the language in Chapter 6 is extended with non-atomic data types, e.g. arrays, lists and higher-order data? To what extent can an offline system with a terminating BTA simulate the accuracy of an online system in such an extended language?

Relational algebra optimization The last four theorems in Chapter 9 can clearly be generalized so that they refer to an abstract aggregation function with certain properties, rather than a concrete aggregation function. It would also be nice to formulate the whole optimization process of that chapter as an algorithm. Furthermore, the benefits obtained by applying my optimizations could probably be expressed as a function of certain measures on the tables in the underlying database.

Domain stability and community In the introduction (Chapter 1), I briefly noted that there may not be a general consensus as to *what* constitutes a problem domain, and as such is fit for being described by a domain-specific language. There is no simple criterion.

In my own application (Chapter 7), it became clear that the problem domain was not stable – it kept developing throughout the project. Take good notice: it was not only my understanding of the problem domain that developed, it really was the domain itself that was chaging. This shouldn't really be a surprise. After all, my problem domain was defined by a program, and programs are artefacts that *do* tend evolve over time. As the *Visanti* programmers were learning more about their product and its use, and as customers came up with new requirements for the program, the problem domain defined be the program evolved. In other words, the problem domain I worked with was in a very real sense *socially constructed* and depended on the *Visanti* programmers and the customers.

Now, it can be argued that this perspective applies to anything that has been suggested as a problem domain, be it 3D animation, device drivers or context-free parsing[1]. It seems that the extreme instability of my problem domain simply stems from the fact that the number of people that contribute to defining it is very small (a few programmers, a few customers). At the other end of the "stability-spectre" you may find e.g. the domain of problems relating to solving partial differential equations. The concepts and methods from that domain are decades old – even centuries old – and are known to hundreds of thousands of people. If you wrote a textbook on partial differential equations and discarded all existing notation to invent your own, no one would accept your exposition. When I wrote Chapter 7 on my problem domain, I was free to invent new names, notations and perspectives, because none of these had properly stabilized.

The point of these observations is that when a problem domain is not stable, implementing a domain-specific language for it is probably risky. If concepts or methods change, the language must be reinvented, which is likely to be a major effort. If you want to avoid this risk, the object to study is not the problem domain itself as given by e.g. a set of example problems, but the people that define the problem domain by having a stable frame of reference for your example problems. Rather than *domain-specific*, our little languages are *community-specific*.

The future work direction implied by these thoughts is to answer the following questions:

1. Does it matter? I.e., does focussing on a community rather than a set of problems have any practical consequences for how we design and implements little languages?

2. If so, can programming language techniques and tools be made to support this alternative focus?

Some discussions on DSL design methodologies ([CE00], Chapter 2) acknowlegde the need for analysing "stakeholder" *interest*, *goals* and *requirements*, but they do not directly address domain stability or provide clear answers to the above questions.

[1]The example of problem domains in this paragraph can all be found in [vDKV00].

Bibliography

[AC94] J. Michael Ashley and Charles Consel.
Fixpoint computation for polyvariant static analyses of higher-order applicative programs.
ACM TOPLAS, 16(5):1431–1448, 1994.

[And93] Lars Ole Andersen.
Binding-time analysis and the taming of C pointers.
In *Proc. of the Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 47–58. ACM Press, 1993.

[And94] L. O. Andersen.
Program Analysis and Specialization for the C Programming Language.
PhD thesis, DIKU, University of Copenhagen, May 1994.
(DIKU report 94/19).

[App98] Andrew W. Appel.
Modern Compiler Implementation in Java.
Cambridge University Press, Cambridge, UK, January 1998.

[Arm97] Joe Armstrong.
The development of Erlang.
In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP-97)*, volume 32,8 of *ACM SIGPLAN Notices*, pages 196–203, New York, June 9–11 1997. ACM Press.

[Arm00] Phillip G. Armour.
The five orders of ignorance.
Communications of the ACM, 43(10):17–20, 2000.

[Arm01] Phillip Armour.
The business of software: Zeppelins and jet planes: a metaphor for modern software projects.
Communications of the ACM, 44(10):13–15, October 2001.

[Asa99] Kenichi Asai.
Binding-time analysis for both static and dynamic expressions.
In A. Cortesi and G. Filé, editors, *Static Analysis. Proceedings*, LNCS 1694, pages 117–133. Springer-Verlag, 1999.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers Principles, Techniques, and Tools.
Addison Wesley, 1986.

[AVWW] J. Armstrong, R. Virding, M. Wikström, and M. Williams.
Concurrent Programming in Erlang.
Prentice-Hall, Englewood Cliffs, NJ.

[BD91] Anders Bondorf and Olivier Danvy.
Automatic autoprojection of recursive equations with global variables and abstract data types.
Science of Computer Programming, 16:151–195, 1991.

[BDK$^+$96] M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. A. van der Meulen.
Industrial applications of ASF+SDF.
In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[Ben86] Jon Louis Bentley.
Programming pearls: Little languages.
Communications of the ACM, 29(8):711–721, August 1986.
Description of the *pic* language.

[BHOS76] Lennart Beckman, Anders Haraldson, Östen Oskarsson, and Erik Sandewall.
A partial evaluator and its use as a programming tool.
Artificial Intelligence, 7:319–357, 1976.

[Bon92] Anders Bondorf.
Improving binding times without explicit CPS-conversion.
In *ACM Conference on Lisp and Functional Programming*, pages 1–10. ACM Press, 1992.

[Bon93] Anders Bondorf.
Similix 5.0 Manual.
DIKU, University of Copenhagen, Denmark, 1993.
Included in the Similix distribution (http://www.diku.dk/forskning/topps/activities/similix.html), 82 pages.

[Bra61] Harvey Bratman.
An alternate form of the ”uncol diagram”.
Communications of the ACM, 4(3):142, 1961.

[BSV03] Claus Brabrand, Michael Schwartzbach, and Mads Vanggaard.
The metafront system: Extensible parsing and transformation.
In *Proceedings of the Third Workshop on Language Descriptions, Tools and Applications (LDTA 2003)*, Warsaw, Poland, April 2003.

[Bul84] Mikhail A. Bulyonkov.
Polyvariant mixed computation for analyzer programs.
Acta Informatica, 21:473–484, 1984.

[Bul93] Mikhail A. Bulyonkov.
Extracting polyvariant binding time analysis from polyvariant specializer.
In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 59–65. ACM Press, 1993.

[CD91a] Charles Consel and Olivier Danvy.
For a better support of static data flow.
In J. Hughes, editor, *Functional Programming Languages and Computer Architecture. Proceedings*, LNCS 523, pages 496–519. Springer-Verlag, 1991.

[CD91b] Charles Consel and Olivier Danvy.
Static and dynamic semantics processing.
In *ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pages 14–24. ACM, January 1991.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker.
Generative Programming: Methods, Tools, and Applications.
Addison-Wesley, 2000.

[CG00] Luca Cardelli and Andrew D. Gordon.
Mobile ambients.
Theoretical Computer Science, 240(1):177–213, June 2000.

[CGJ$^+$00] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding.
Core Erlang 1.0 language specification.
Technical Report 2000-030, Department of Information Technology, Uppsala University, November 2000.

[CGL00] Niels H. Christensen, Robert Glück, and Søren Laursen.

Binding-time analysis in partial evaluation: one size does not fit all.
In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Perspectives of System Informatics. Proceedings*, LNCS 1755, pages 80–92. Springer-Verlag, 2000.

[CHN$^+$96] Charles Consel, Luke Hornof, François Nöel, Jacques Noyé, and Eugen Nicolae Volanschi.
A uniform approach for compile-time and run-time specialization.
In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation. Proceedings*, LNCS 1110, pages 54–72. Springer-Verlag, 1996.

[CK95] Charles Consel and Siau-Cheng Khoo.
On-line and off-line partial evaluation: semantic specifications and correctness proofs.
Journal of Functional Programming, 5(4):461–500, 1995.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee.
Symbolic Logic and Mechanical Theorem Proving, chapter 10.9 The Specialization of Programs, pages 228–231.
Computer Science and Applied Mathematics. Academic Press, New York, London, 1973.

[CM98] Charles Consel and Renaud Marlet.
Architecturing software using: A methodology for language development.
Lecture Notes in Computer Science, 1490:170–??, 1998.

[Con93] Charles Consel.
Polyvariant binding-time analysis for applicative languages.
In *Workshop on Partial Evaluation and Semantics-Based program Manipulation*, pages 66–77. ACM Press, 1993.

[CØV02] Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter.
Generative programming.
Lecture Notes in Computer Science, 2548:15–29, 2002.

[CS94] Surajit Chaudhuri and Kyuseok Shim.
Including group-by in query optimization.
In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 354–366, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

[DFG98] Mads Dam, Lars-Åke Fredlund, and Dilian Gurov.
Toward parametric verification of open distributed systems.

Lecture Notes in Computer Science, 1536:150–185, 1998.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors.
Language Prototyping: An Algebraic Specification Approach, volume 5 of *AMAST Series in Computing*.
World Scientific Publishing Co., 1996.

[Die01] Suzanne W. Dietrich.
Understanding Relational Database Query Languages.
Prentice-Hall, 2001.

[EFdM00] Conal Elliott, Sigbjorn Finne, and Oege de Moor.
Compiling embedded languages.
In *SAIG*, pages 9–27, 2000.

[EHK96] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek.
'C: A language for high-level, efficient, and machine-independent dynamic code generation.
In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144. ACM SIGACT and SIGPLAN, ACM Press, 1996.

[Ers77] Andrei P. Ershov.
On the partial computation principle.
Information Processing Letters, 6(2):38–41, 1977.

[ES70] Jay Earley and Howard Sturgis.
A formalism for translator interactions.
Communications of the ACM, 13(10):607–617, 1970.

[FGS97] Markus P.J. Fromherz, Vineet Gupta, and Vijay A. Saraswat.
Cc - a generic framework for domain specific languages.
In *Proc. 1st ACM-SIGPLAN Workshop on Domain-Specific-Languages, DSL '97, Paris, France, January 18, 1997*. Technical Report, University of Illinois at Urbana-Champaign, 1997.

[FN88] Yoshihiko Futamura and Kenroku Nogi.
Generalized partial computation.
In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[FNP97] Rickard E. Faith, Lars S. Nyland, and Jan F. Prins.
KHEPERA: A system for rapid implementation of domain specific languages.

In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 243–256, Berkeley, October 15–17 1997. USENIX Association.

[Fou98] Cédric Fournet.
The Join-Calculus: a Calculus for Distributed Mobile Programming.
PhD thesis, Ecole Polytechnique, 1998.

[Fut71] Yoshihiko Futamura.
Partial evaluation of computing process – an approach to a compiler-compiler.
Systems, Computers, Controls, 2(5):45–50, 1971.
Reprinted in Higher-Order and Symbolic Computation, 12(4): 381–391, 1999.

[GHS02] Jens Christian Godskesen, Thomas Hildebrandt, and Vladimiro Sassone.
A calculus of mobile resources.
In Luboš Brim, Petr Jančar, Mojmír Křetinský, and Antonín Kučera, editors, *CONCUR 2002: Concurrency Theory (13th International Conference, Brno, Czech Republic)*, volume 2421 of *LNCS*, pages 272–287. Springer, August 2002.

[GJ91] Carsten K. Gomard and Neil D. Jones.
Compiler generation by partial evaluation: a case study.
Structured Programming, 12:123–144, 1991.

[GJ94] Robert Glück and Jesper Jørgensen.
Generating optimizing specializers.
In *IEEE International Conference on Computer Languages*, pages 183–194. IEEE Computer Society Press, 1994.

[GJ96] Arne J. Glenstrup and Neil D. Jones.
BTA algorithms to ensure termination of off-line partial evaluation.
In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, LNCS 1181, pages 273–284. Springer-Verlag, 1996.

[GK93] Robert Glück and Andrei V. Klimov.
Occam's razor in metacomputation: the notion of a perfect process tree.
In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Static Analysis. Proceedings*, LNCS 724, pages 112–123. Springer-Verlag, 1993.

[Glü02] Robert Glück.

Jones optimality, binding-time improvements, and the strength of program specializers.
In *Proceedings of the Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 9–19. ACM Press, 2002.

[GM96] M. Gengler and M. Martel.
Self-applicable partial evaluation for the pi-calculus.
Technical report, Ecole Normale Superieure de Lyon, Laboratoire de l'Informatique du Parallelisme, 1996.

[GMS99] Arne Glenstrup, Henning Makholm, and Jens Peter Secher.
C-Mix: specialization of C programs.
In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation. Practice and Theory*, LNCS 1706, pages 108–153. Springer-Verlag, 1999.

[Gro03] Austin Grossman, editor.
Post Mortems from Game Developer.
CMP Books, 2003.

[GS94] Robert Glück and Morten Heine Sørensen.
Partial deduction and driving are equivalent.
In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings*, LNCS 844, pages 165–181. Springer-Verlag, 1994.

[GS96] Robert Glück and Morten Heine Sørensen.
A roadmap to metacomputation by supercompilation.
In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation. Proceedings*, LNCS 1110, pages 137–160. Springer-Verlag, 1996.

[Hat99] John Hatcliff.
An introduction to online and offline partial evaluation using a simple flowchart language.
In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation. Practice and Theory*, LNCS 1706, pages 20–82. Springer-Verlag, 1999.

[HDL98] John Hatcliff, Matthew Dwyer, and Shawn Laubach.
Staging analysis using abstraction-based program specialization.
In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming. Proceedings*, LNCS 1490, pages 134–151. Springer-Verlag, 1998.

[HKY96] H. Hosoya, N. Kobayashi, and A. Yonezawa.
Partial evaluation scheme for concurrent languages and its correctness.
In L. Bougé et al., editors, *Euro-Par'96 - Parallel Processing, Lyon, France. (Lecture Notes in Computer Science, vol. 1123)*, pages 625–632. Berlin: Springer-Verlag, 1996.

[HM02] Elliotte Rusty Harold and W. Scott Means.
XML in a nutshell.
In a nutshell. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, second edition, 2002.

[HN99] Luke Hornof and Jacques Noyé.
Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity.
Theoretical Computer Science, 248(1–2):3–27, 1999.

[Huc99] Frank Huch.
Verification of erlang programs using abstract interpretation and model checking.
In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP-99)*, volume 34.9 of *ACM Sigplan Notices*, pages 261–272, N.Y., September 27–29 1999. ACM Press.

[Hud98a] Paul Hudak.
Handbook of Programming Languages, volume III, chapter 3.
Macmillan Technical, 1998.

[Hud98b] Paul Hudak.
Modular domain specific languages and tools.
In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[JG02] Neil D. Jones and Arne Glenstrup.
Program generation, termination, and binding-time analysis.
In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering. Proceedings*, LNCS 2487, pages 1–31. Springer-Verlag, 2002.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft.
Partial Evaluation and Automatic Program Generation.

Prentice Hall International, International Series in Computer Science, June 1993.
ISBN number 0-13-020249-5 (pbk).

[Jon88] Neil D. Jones.
Automatic program specialization: a re-examination from basic principles.
In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[Jon97] Neil D. Jones.
Computability and Complexity from a Programming Perspective.
Foundations of Computing. MIT Press, Boston, London, 1 edition, 1997.

[JSS85] Neil D. Jones, Peter Sestoft, and Harald Søndergaard.
An experiment in partial evaluation: the generation of a compiler generator.
In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, LNCS 202, pages 124–140. Springer-Verlag, 1985.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard.
Mix: a self-applicable partial evaluator for experiments in compiler generation.
LISP and Symbolic Computation, 2(1):9–50, 1989.

[Kam] S. Kamin.
Building program generators the easy way (extended abstract).
Technical report, University of Illinois at Urbana-Champaign.

[Kam98] S. Kamin.
Research on domain-specific embedded languages and program generators.
Electronic Notes in Theoretical Computer Science, 1998.

[KKZG95] Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöchling, and Robert Glück.
Fortran program specialization.
SIGPLAN Notices, 30(4):61–70, 1995.

[KMB$^+$96] Richard B. Kieburtz, Laura McKinney, Jeffery M. Bell, James Hook, Alex Kotov, Jeffery Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, and Lisa Walton.
A software engineering experiment in software component generation.

In *Proceedings of the 18th International Conference on Software Engineering*, pages 542–553. IEEE Computer Society Press, 1996.

[KST98] P. Kutter, D. Schweizer, and L. Thiele.
Integrating domain specific language design in the software life cycle.
In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*, volume 1641 of *LNCS*, pages 196–212, Boppard, Germany, October 1998.

[Lan66] P. J. Landin.
The next 700 programming languages.
Communications of the ACM, 9(3):157–164, March 1966.
Originally presented at the Proceedings of the ACM Programming Language and Pragmatics Conference, August 8–12, 1965.

[LD94] Julia L. Lawall and Olivier Danvy.
Continuation-based partial evaluation.
In *ACM Conference on Lisp and Functional Programming*, pages 227–238. ACM Press, 1994.

[LM99] Daan Leijen and Erik Meijer.
Domain-specific embedded compilers.
In USENIX, editor, *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL '99), October 3–5, 1999, Austin, Texas, USA*, pages 109–122, Berkeley, CA, USA, 1999. USENIX.

[LS91] John W. Lloyd and J. C. Shepherdson.
Partial evaluation in logic programming.
Journal of Logic Programming, 11(3–4):217–242, 1991.

[Mar00a] Renaud Marlet.
Tempo, a program specializer for C.
SIGPLAN Notices, 35(7):76–77, 2000.

[Mar00b] Matthieu Martel.
Analyse Statique et Evaluation Partielle de Systèmes de Processus Mobiles.
PhD thesis, Université d'Aix-Marseille II, 2000.

[Mey91] Uwe Meyer.
Techniques for partial evaluation of imperative languages.
In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105. ACM Press, 1991.

[Mey99] Uwe Meyer.
Correctness of online partial evaluation for a Pascal-like language.
Science of Computer Programming, 34(1):55–73, 1999.

[MG97] M. Marinescu and B. Goldberg.
Partial-evaluation techniques for concurrent programs.
In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 47–62, New York, June 12–13 1997. ACM Press.

[MG00] Matthieu Martel and Marc Gengler.
Communication topology analysis for concurrent programs.
In *SPIN'2000*, volume 1885 of *LNCS*, pages 265–286. Springer, 2000.

[MG01] Matthieu Martel and Marc Gengler.
Partial evaluation of concurrent programs.
Lecture Notes in Computer Science, 2150:504–514, 2001.

[Mil99] Robin Milner.
Communicating and Mobile Systems: The π Calculus.
Cambridge University Press, Cambridge, England, 1999.

[Mog02] Torben Mogensen.
Roll: A language for specifying die-rolls.
In veronica Dahl and Philip Wadler, editors, *PADL 2003*, volume 2562 of *LNCS*, pages 145–159. Springer, 2002.

[Mog03] Torben Æ. Mogensen.
Linear types for cashflow reengineering.
In *Perspectives of System Informatics '03*, LNCS. Springer, 2003.

[MWW02] S. Mauw, W. Wiersma, and T. Willemse.
Language-driven system design.
In *HICSS35, Proceedings of the Hawaii International Conference on System Sciences, minitrack on Domain-Specic Languages for Software Engineering, Hawaii*, January 2002.
To appear.

[MY01] Hidehiko Masuhara and Akinori Yonezawa.
Run-time bytecode specialization: a portable approach to generating optimized specialized code.
In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects. Proceedings*, LNCS 2053, pages 138–154. Springer-Verlag, 2001.

[NAOP00] Lloyd H. Nakatani, Mark A. Ardis, Robert G. Olsen, and Paul M. Pontrelli.
Jargons for domain engineering.
ACM SIGPLAN Notices, 35(1):15–24, January 2000.

[NJ97] L. Nakatani and M. Jones.
Jargons and infocentrism.
In *Proceedings of the first ACM SIGPLAN Workshop on Domain-Specific Languages*, pages 59–74, 1997.

[Orw99] Jon Orwant.
EGGG: The Extensible Graphical Game Generator.
PhD thesis, MIT, 1999.

[Orw00] J. Orwant.
EGGG: Automated programming for game generation.
IBM Systems Journal, 39(3/4):782–794, 2000.

[Ous94] John K. Ousterhout.
Tcl and the Tk Toolkit.
Addison Wesley, 1994.

[Ous98] John K. Ousterhout.
Scripting: Higher-level programming for the 21st century.
Computer, 31(3):23–30, 1998.

[PK97] Peter Pfahler and Uwe Kastens.
Language design and implementation by selection.
In *Proc. 1st ACM-SIGPLAN Workshop on Domain-Specific-Languages, DSL '97, Paris, France, January 18, 1997*, pages 97–108. Technical Report, University of Illinois at Urbana-Champaign, 1997.

[RG92] Bernhard Rytz and Marc Gengler.
A polyvariant binding time analysis.
In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–28. Yale University, Dept. of Computer Science, 1992.

[RM01] L. Réveillère and G. Muller.
Improving driver robustness: an evaluation of the devil approach.
In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*, pages 131–140, Washington - Brussels - Tokyo, July 2001. IEEE.

[Rom88] Sergei A. Romanenko.

A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure.
In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.

[Ruf93] Eric Ruf.
Topics in online partial evaluation.
Technical Report CSL-TR-93-563, Stanford University, Computer Systems Laboratory, 1993.

[RW93] Eric Ruf and Daniel Weise.
On the specialization of online program specializers.
Journal of Functional Programming, 3(3):251–281, 1993.

[Sch01] Ulrik Pagh Schultz.
Partial evaluation for class-based object-oriented languages.
In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects. Proceedings*, LNCS 2053, pages 173–197. Springer-Verlag, 2001.

[SeaBP99] Tim Sheard, Zine el-abidine Benaissa, and Emir Pasalic.
DSL implementation using staging and monads.
In USENIX, editor, *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL '99), October 3–5, 1999, Austin, Texas, USA*, pages 81–94, Berkeley, CA, USA, 1999. USENIX.

[SG97a] Diomidis Spinellis and V. Guruprasad.
Lightweight languages as software engineering tools.
In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 67–76, Berkeley, October 15–17 1997. USENIX Association.

[SG97b] James M. Stichnoth and Thomas Gross.
Code composition as an implementation language for compilers.
In USENIX, editor, *Proceedings of the Conference on Domain-Specific Languages, October 15–17, 1997, Santa Barbara, California*, pages 119–132, Berkeley, CA, USA, 1997. USENIX.

[SGJ96] Morten Heine Sørensen, Robert Glück, and Neil D. Jones.
A positive supercompiler.
Journal of Functional Programming, 6(6):811–838, 1996.

[SGT96] Michael Sperber, Robert Glück, and Peter Thiemann.
Bootstrapping higher-order program transformers from interpreters.

In K. M. George, J. H. Carroll, D. Oppenheim, and J. Hightower, editors, *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 408–413. ACM Press, 1996.

[Shi96] O. Shivers.
A Universal Scripting Framework or Lambda: The Ultimate “little language”.
Lecture Notes in Computer Science, 1179:254–265, 1996.

[SK00] Eijiro Sumii and Naoki Kobayashi.
Online-and-offline partial evaluation: a mixed approach.
In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–21. ACM Press, 2000.

[TBS98] W. Taha, Z. Benaissa, and T. Sheard.
Multi-stage programming: Axiomatization and type safety.
In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 1998.

[Thi98] Scott Thibault.
Langage Dédiés: Conception, Implémentation et Application.
Thèse de doctorat, Université de Rennes 1, France, October 1998.

[TMC99] S. A. Thibault, R. Marlet, and C. Consel.
Domain-Specific Languages: From Design to Implementation Application to Video Device Drivers Generation.
IEEE Transactions on Software Engineering, 25(3):363–377, May 1999.

[TS96] Peter Thiemann and Michael Sperber.
Polyvariant expansion and compiler generators.
In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, LNCS 1181, pages 285–296. Springer-Verlag, 1996.

[Tur36] Alan M. Turing.
On computable numbers, with an application to the Entscheidungsproblem.
Proc. London Math. Soc., 2(42):230–265, 1936.

[Tur86] Valentin F. Turchin.
The concept of a supercompiler.
ACM Transactions on Programming Languages and Systems, 8(3):292–325, 1986.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser.
Domain-specific languages: An annotated bibliography.

http://www.cwi.nl/ arie/papers/dslbib/dslbib.html, 2000.

[Wad95] P. Wadler.
Monads for functional programming.
Lecture Notes in Computer Science, 925:24–54, 1995.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman.
Automatic online partial evaluation.
In *Functional Programming Languages and Computer Architectures. Proceedings*, LNCS 523, pages 165–191. Springer-Verlag, 1991.

[Wei] D. Weiss.
Defining families: The commonality analysis.
submitted to IEEE Transactions on Software Engineering.