CONCURRENT TREE SPACE TRANSFORMATION IN THE AARDAPPEL PROGRAMMING LANGUAGE

By
Wouter van Oortmerssen Candidate
M.A.

A thesis submitted for the degree of
Doctor of Philosophy

Department of Electronics and Computer Science,
University of Southampton,
United Kingdom.

July 2000

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE

ELECTRONICS AND COMPUTER SCIENCE DEPARTMENT

Doctor of Philosophy

Concurrent Tree Space Transformation
In The Aardappel Programming Language

by Wouter van Oortmerssen Candidate

It is perceived that one of the biggest problems in maintaining software quality is the above linear growth in complexity compared to the size of a program, resulting in the programmer's cognitive loss of an overview and ultimately the degradation of the quality of the software. This thesis tries to counter that by introducing a language with a new sharing model that makes dependencies in a program explicit at the language level, and local to one specific language construct (the tree space). We introduce the language which is based on tree rewriting and Linda style tuple spaces and comes with a graphical programming notation. We discuss the worth of its design, precisely specify it using a formal semantics, and report on experience with the model using a real world implementation.

Acknowledgements

I would like to thank my supervisor, Michael Butler, for his patience with me and his insights into the formal semantics of the language. I would like to thank the many people at the University of Southampton for their help and inspiration during these 3 years, in particular Don Cruickshank, Hugh Glaser, Pieter Hartel, Stuart Maclean, Luc Moreau, Manfred Münch, Leonid Mikhailov, Mark Longley and Charlie Boardman. Thanks to Alan Wood for his input, and Tim Rennie for testing AardEd. Special thanks go to Rob Verver for years of in-depth discussion on language design, feedback and inspiration, without which I'm sure this research would never have happened. And most importantly, thanks to my family for their loving care and support through these years.

Wouter van Oortmerssen

Contents

Chapter 1

Introduction

1.1 Scope of this research

In very general terms, the goals of this research could be described as: "reducing the complexity of creating and maintaining software, and improving software quality". The actual contribution this research makes will be very modest, however, because the problems associated with software creation are so huge and such an intrinsic part of what software is, that it is doubtful that a technique that will "solve" the problems will ever exist. Most research so far in this area can be classified as introducing a new technique to improve a small aspect of software creation, that can nevertheless have a significant impact, and this thesis tries to do exactly that.

There are many angles of attack to contribute to the above goal, which correspond to different schools of research in the area of computer science, for example software engineering practices and tools, formal methods, metrics, language design, and programming environments. It is rather pointless to argue about which of these is the better approach, or which will make a bigger step towards our goal. It's safe to assume progress in *all* of these areas is necessary for the overall situation to improve, and this particular research tries to make a contribution in the language design field (and, to a lesser extend, programming environments).

1.2 The problem

It is not easy to define the problem exactly, but at least the consequences of it are well known: for the user of the software these are *bugs*, and additionally for the programmer

maintenance is increasingly difficult. There are many circumstances that can lead to bugs and difficult to maintain code, but one in particular forces the issue: when the complexity of a program grows to such an extent that the programmer *loses track* of finer details of its structure. Alternatively, one could say that there is a struggle between what complexity a programmer can cognitively encompass, and the structure of a program. Once the boundary of the programmer's cognitive capacity is broken, maintenance becomes more and more difficult, bugs are introduced more easily, and in general quality goes down rapidly.

There are many features in programming languages that contribute to overall structural complexity, but in this thesis we will summarize most of them as *dependencies*. We define a dependency as *any relation of a static or run-time entity in the program with another*[1]. The problem with dependancies is that they can grow *explosively* relative to the number of modifications of a program, in the worst case. To see how this can happen, imagine all dependancies in a program as one big graph. For a certain modifiable item, there may be `n` dependancies on it. If I now add code that depends on this item, that code will also depend on all dependents of this item, because how they deal with the item can affect my code. So effectively adding a single unit of code can add `n+1` dependancies to the graph, which is far from linear. With a great number of modifications this can quickly cause the complexity of the dependency graph to explode, and stress the cognitive capacity of a programmer beyond its limits.

In the design of most programming languages, the importance of these dependencies is secondary to the semantics of the language features. In fact, most languages contain no language features to help you get a grip on the dependency graph, it is invisibly layered within a program. The main goal of this research has been to counter this, and come up with a language design which gives the programmer an explicit and clearly visible view of the dependancies in a program, at the language level. It does this by introducing a new *sharing model* to deal with dependancies between modifiable dynamic data structures.

Aardappel's [53] sharing model attempts to merge the best properties of the two basic models (the safety of functional programming, and the flexibility of imperative programming) by making the dependency relation between a runtime data structure and its *observers* explicit as a language feature, the *tree space*, and enforcing linear (unshared) access to the tree space from outside it.

[1] An example of a dynamic dependancy would be that of a data structure and another data structure that refers to it (using a pointer), a static one would be between the definition and a use of variable.

1.3 Thesis structure

The rest of this thesis is structured as follows:

Chapter 2 provides a background to all topics that are the building blocks of the Aardappel, and to some extent also shows problems with existing language design.

Chapter 3 describes the Aardappel programming language syntax and semantics informally, and provides some simple examples.

Chapter 4 discusses the design in more depth, and explains exactly what advantages they give, what problem they tackle etc.

Chapter 5 contains a formal semantics of the language, and discusses some of the more subtle properties of the semantics.

Chapter 6 looks at the implementation of the language, the technologies used, and experience in using the programming environment.

Chapter 7 presents a case study, and compares the way Aardappel deals with sharing with two well known proponents of imperative and functional programming.

Chapter 8 discusses related work, and compares with the Aardappel approach.

Chapter 9 concludes.

Chapter 2

Background

In this chapter we will look at four programming language topics which form the groundwork upon which the Aardappel is designed. *Tree rewriting* and *Linda*, because they are the basis of Aardappel's sequential and concurrent computation respectively. the section on *sharing* gives some insights to later value Aardappel's sharing model, and finally *graphical languages* provides some context on the topic of Aardappel's syntax.

2.1 Tree rewriting

Tree rewriting is a model of computation which is used in a variety of contexts within computer science, for example in semantic specification and compiler implementation (as in e.g. [34]), mostly thanks to its great simplicity[1]. In a more restricted sense tree rewriting can be seen as a class of (declarative) programming languages [51] and that is the perspective we'll take in this thesis. For some reason, tree rewriting as a programming language has seen few precedents (see section 8.1 for references).

2.1.1 Rules

Informally tree rewriting does just that: it rewrites trees into other trees. specifying how a tree is rewritten is done in the form of a *rule*. When a rule is applied, in what order, and when this rewriting process terminates differs across the class of possible tree rewriting languages, as we'll see below.

To aid the discussion, we introduce a generic tree rewriting language, with the following syntax:

[1]Much like lambda calculus, because its own semantics is so well defined and intuitive, it is very suitable for expressing others.

$$
\begin{array}{lcl}
\textit{rule} & = & \textit{tree}\ \text{"="}\ \textit{tree} \\
\textit{tree} & = & \textit{atom}\ \text{"("}\ \textit{children}\ \text{")"} \\
 & | & \textit{atom} \\
 & | & \textit{var} \\
 & | & \textit{int} \\
\textit{children} & = & \textit{children}\ \text{","}\ \textit{tree} \\
 & | & \textit{tree}
\end{array}
$$

Integers, variables and atoms are defined in the usual way as numbers and identifiers, and integers are just a special case of atoms added here to make things interesting. A tree is an atom followed by a list of children, in any nested fashion. A rule is specified using two trees, a pattern and an expression, also called the left hand side and right hand side.

A program is specified by a set of rules, and a starting expression (a tree). The meaning of this program (the result) is another tree, obtained by repeatedly applying rules to the starting tree until no rule applies any more (if this situation applies, the tree is said to be in *normal form*). To be able to apply a rule to tree, the tree and the pattern part have to *match*. Two trees match if they are structurally equivalent, except that variables (placeholders) in the pattern (rule left hand side) are considered a match for any value. This allows a rule to abstract over the sort of trees it can apply to and not just match on concrete values. If the tree matches with the pattern, the actual rewriting step can be performed as follows: the tree that was matched is replaced by the tree in the right hand side of the rule, except that any variables that occur in it that also occur in the left hand side are replaced by the values they matched with. A variable on the right hand side that doesn't occur on the left hand side is called a free variable and isn't allowed in tree rewriting.

A simple example would be rewriting the tree `a(b(1),f(2))` using the following 3 rules:

```
a(b(x),y) = c(y,d(x))
d(y) = e(y)
c(x,e(y)) = a(x,y)
```

`x` and `y` in these rules are place holders, the rest are concrete values. At the start, only the first rule will match, applying the rule will give the bindings `x = 1` and `y = f(2)`, and thus construct the new tree `c(f(2),d(1))`. This will make only the second rule applicable, which similarly produces the binding `y = 1`, and rewrites the tree to `c(f(2),e(1))`. Note that only now the final rule is applicable, which rewrites the tree

to `a(f(2),1)`, using `x = f(2)` and `y = 1`. This tree now has no rules applicable to it anymore, so it is in normal form.

Now we know how to perform rewrite steps for a tree, but there are still two questions that remain: if we can potentially rewrite multiple parts of a tree, which do we rewrite first? and what if multiple rules can be applied to the same part of the tree? The answers to these question tell us what sort of tree rewriting language we're dealing with, and determine *evaluation order* and *rule selection order* respectively.

2.1.2 Evaluation order and termination

If we are rewriting a tree, there can be potentially multiple subtrees that have rules that apply to them. Which subtree we rewrite first matters, because a rewrite step replaces the subtree with a potentially completely different value, which affects whether neighbouring subtrees can still be rewritten, and thus the overall outcome. Worse, because it influences applicability of rules, it changes the termination properties of a program. Finally, evaluation order affects the efficiency of a program, up to orders of magnitude in difference. The following are some of the possible strategies:

- Normal order evaluation (leftmost outermost). This is considered by many the most "natural" evaluation strategy for tree rewriting[2]: the subtree that gets rewritten first is the outermost subtree that can be rewritten (has a rule applicable to it). This is most natural because if you rewrite a tree starting at the top, you can't ruin rewriting opportunities for trees that encapsulate you, i.e. it works correctly with context sensitive rewrites (see below). Also, because it is conservative towards rewriting opportunities, its termination properties are what you intuitively expect. The downside is that its semantics specify an evaluation that potentially gives a program a much higher complexity (sometimes orders of magnitude) than its algorithm traditionally specifies. This is because the rewriting of trees bound to placeholders occurring more than once on the right hand side of a rule can't be shared: depending on their parent trees, they may be rewritten using completely different sets of rules. Not sharing them however can mean that common parts are evaluated more than once, with potentially exponential consequences (program analysis can remove a lot of inefficiencies but so far without being able to guarantee it [61] and [62]).

[2]You'll find that if you come across a text that talks about tree rewriting without specifying its evaluation order, they implicitly assume normal order evaluation.

- Eager evaluation (innermost). This strategy corresponds closely to what most people are used to from other languages (imperative and strict functional ones): the subtree that gets rewritten is the innermost subtree that can be rewritten. The obvious attractions of this strategy is that implementations can be simple and fast, and that complexity of evaluation is very predictable. The downside is that it has worse termination properties than normal order evaluation, and doesn't allow for the same natural style of programming (it enforces a more functional style which emerges from the fact that innermost evaluation causes all children of a tree to which a rule is being applied to be in normal form: this makes the outermost atom in a rule much like a function and the others much like data types).
- Non-deterministic evaluation. This one is more of a theoretical interest than practically applicable as programming language: the subtree that gets rewritten is the one that leads to a terminating sequence of rewrite steps. This potentially results in multiple normal forms, but it is not generally possibly to know exactly how many there are (many paths won't terminate).

As an example of termination properties of different evaluation schemes, consider the following rules:

```
a(b(x)) = x
b(x) = a(b(x))
```

In a tree rewriting language with normal order evaluation, we'd expect trees like `b(1)` and `a(b(1))` to evaluate to `1`, and if you look at the rules that is what you'd expect of pattern matching. But in an eager language, rewriting would never terminate, and both trees would generate an infinite sequence of `a(a(a(a(a(...)))))`. Eager evaluation will give precedence to rewriting inner trees, and in this case it can do that. It is also possible to construct an example that will terminate using eager evaluation and won't using normal order evaluation:

```
a(b(x)) = x
a(x) = a(c(x))
c(x) = b(x)
```

The tree `a(1)` will evaluate to `1` using eager evaluation or to the infinite sequence `a(c(c(c(...1...))))` in the normal order case. This relies on an inner tree being rewritten to trigger the terminating version of a rule, and as such a bit more contrived than the first example.

2.1.3 *Rule selection order*

The second axis along which we can order possible tree rewriting languages is what strategy is used to resolve the ambiguity that arises when the number of rules that can be applied to a tree is greater than one. The influence of this property on the semantics of the language is not as strong as evaluation order, but does have an effect on how the programmer writes programs, as we'll see in later chapters. Amongst possible strategies are the following:

- Source code order. This is the simplest strategy, and determines that the rule to be picked is the first one that applies in the program. This one is simplest to implement, and at least for small programs, also simplest to use.
- Uniqueness requirement. This requires the programmer to ensure that no rules overlap, and no tree can cause an ambiguous situation. To cater for this such a language will usually allow for negative guards that make sure the current rule doesn't accept trees which match onto similar rules. The advantages of this are that programs will contain fewer surprises as it's easier to spot which type of trees will trigger which rules. However the added precision in specification comes at a cost: programs are cumbersome to write and often these negative guards are so obvious (seem superfluous) that they obscure the actual code.
- Specificity ordering. This defines an order between any two rules depending on how *specific* they are. The general idea is that the more complex a pattern of a rule, the more specific it is, and the more likely it should be given precedence in case of an ambiguity[3]. This is a very clever strategy as it gives greater freedom in program construction and enables additional programming techniques (as we'll see later), downside is that which type of trees will trigger which rules becomes even less obvious.
- Non-deterministic selection. Again only out of theoretical interest: the rule to select is the one that leads to a terminating evaluation.

2.1.4 *Dynamic Typing*

As you can see from the grammar above, the only way for a rule to decide if a tree matches is to pattern match on its structure, and a tree not matching on any rules has

[3]There are many ways to define the relation "more specific than" on two trees, depending on which properties you consider more important: number of children, complexity (size) of a subtree, specificity of subtrees left to right or in any order, specificity of basic elements like atoms, integers and variables etc.

a valid meaning too (it is in normal form). Since you can't ever dynamically flag an "illegal argument" or some other form of type error (if a value is illegal for a rule, the rule isn't applicable to it so there is never a clash), you can't do this statically either. It is because of this that a notion of static (strong) typing doesn't make sense in a tree rewriting language. The "type system" of just about every tree rewriting language is that of dynamic typing, similar to languages like Lisp[48], ProLog[21] and Smalltalk[29]. Dynamically typed languages, by their very definition, have parametric polymorphism.

2.1.5 Examples

What familiar examples look like will hardly be surprising to those familiar with functional languages, but it is interesting to look at some evaluation order issues and an example that shows a particular strength of tree rewriting: *context sensitive rewriting*: rewriting trees that trigger different rules depending on what their context is (what parent trees they are part of).

```
append(nil,y) = y
append(cons(h,t),y) = cons(h,append(t,y))
```

A very generic example, in that it doesn't behave any differently for the various evaluation orders examined above. This is because it doesn't repeat any variables more than once on the right hand side (meaning different evaluation strategies don't cause significantly different numbers of evaluation steps), and its written in a functional programming style. As you can see, tree rewriting makes no difference between trees which are meant as functions or data.

The next example is a more typical true tree rewriting style (it only functions as intended under normal order evaluation), using specificity rule selection order:

```
int(x) = cons(push(x),nil)
plus(x,y) = append(x,append(y,cons(add,nil)))
mult(x,y) = append(x,append(y,cons(mul,nil)))

plus(x,int(0)) = x
plus(int(0),x) = x
mult(x,int(1)) = x
mult(int(1),x) = x
```

It is a simplistic code generator with built-in code optimisation: it translates an AST (Abstract Syntax Tree) into a list of stack operations. For example, the expression 3*1+2+0 represented as the AST `plus(plus(mult(int(3),int(1)),int(2)),int(0))` would evaluate to `cons(push(3),cons(push(2),cons(add,nil)))`. It exhibits a different programming style from the functional style. Trying to interpret the above code with eager evaluation won't work: the optimisation rules (the last 4 rules) above would never be used because the rule rewriting `int(x)` would be valid for all integers, and they would never have the chance of being rewritten in the context of other trees. This context sensitive style of programming is unique to tree rewriting[4]. To write the above code in an eager language, one would need to wrap all rules above in an auxiliary tree both at top level on the left hand sides and around most variables on the right hand side.

2.2 Linda

Linda ([17] [13]) is a model of concurrent programming, process creation, and shared memory. An implementation of Linda for an actual language typically adds some new operations to the host language which take care of concurrent communication, synchronisation and creating processes. It is called a *coordination language*, because it is orthogonal to the sequential language which it complements: Linda's task is the *coordination of software ensembles.*

2.2.1 Linda semantics

A Linda program consist of a set of processes and a (conceptually) shared memory called a *tuple space.* Even though Linda is a shared memory model, this says nothing about how it is implemented, and no physical shared memory needs to be present on the machine(s) that are running it[5]. The unit of data contained in the tuple space is the tuple: an unlabeled aggregate of data values. There are two basic operations the processes can perform on the tuple space, reading and writing to it. Writing has very simple semantics: it allows the process to add any tuple to the tuple space, and always succeeds. Reading is a little more complicated: a template (with formal variables and actual values) is specified to denote what class of tuples we want to read from. If a matching tuple is available, formals are bound, and the process continues execution. If

[4]And explains its popularity in situations where complex transformations have to be performed, such as in language implementation & specification.

[5]In fact, the original motivation for the design of Linda was the utilisation of spare computing resources in networks of workstations (NOWs), i.e. a distributed system.

multiple matching tuples are available, it is unspecified which tuple will be picked. If no matching tuples are available, the process trying to read the value will block, and will be blocked until it can read a matching tuple from the tuple space (because another process writes such a tuple). Most incarnations of Linda will feature multiple primitives for reading however, depending on whether the tuple should be removed from the tuple space after a match, and whether a read should block or be a test only (more on this below). Further primitives may exist governing the creation of processes.

The functioning of the tuple space mechanism seems so simplistic, that it is hard to understand just what benefit this approach can give us. It is therefore a good idea to look at some of the properties that emerge from its semantics:

- Communication is not only asynchronous but also anonymous in time and space. Most classical communication mechanisms, most notably message passing, require the sender of a message to know a lot of information about the receiver: its identity, how it can be reached, what it understands, whether it is currently available etc. Linda decouples all this: a communication is anonymous in the sense that a sender has no idea who it is sending the tuple to, and the receiver has no idea who the tuple was from. As long as there is another process participating that can deal with the sort of tuples that are being sent, communication can happen. Anonymity in time opens up another dimension of computational structures: because a communication is, unlike other models, a tangible data structure, it can exist on its own without the need for the sender and/or the receiver to be present. This means a process can read a tuple originating from a process that doesn't exist anymore (this is called *generative communication*), or the whole contents of a tuple space can be stored somewhere to allow a computation to resume at a later time, or serve as a persistency scheme.
- A tuple space transparently supports other communication schemes besides the classical 1 to 1 of message passing: 1 to n, n to 1, and n to n, and all without necessarily planning for it. Processes produce tuples, and to them it is relatively uninteresting how many different processes consume these tuples. Similarly, a process consuming tuples of a certain kind is not interested in how many different processes the tuples originate from. This provides the software equivalent of a *bus*.
- Data structures built from tuples in the tuple space are automatically *distributed data structures*, data structures that can be accessed and modified by multiple processes at the same time without interfering with each other, and without

needing a global lock on the shared data (a monitor). This is an essential property for efficiency in a distributed system.

- Given that a program has been written to sufficiently subdivide the computation and represent the work to be done as tuples, the Linda model naturally schedules to a near optimal degree (i.e. approaching perfect parallel speedup), irrespective of the speeds of the participating hosts, and the sizes of the subtasks involved. "naturally" because it schedules well even in its most naive implementation: it balances the load completely dynamically. This is especially the case if the replicated worker model is adopted: each host will process tuples as per the program, and if a host is heavily loaded, just plainly slower, or is processing a bigger subtask, then it will simply process fewer tuples than the other hosts.

2.2.2 C-Linda

C-Linda ([49]) is a particular implementation of Linda, but definitely the most well know one, and a good starting point for looking at Linda more closely. C-Linda is the C language enriched with Linda primitives, two for writing tuples and four for reading them.

The `out` primitive (written as a C statement, followed by a tuple as all Linda primitives) outputs a tuple to the tuple space. For example:

```
out("search",1);
```

adds a tuple with two elements (a string and an integer) to the tuple space. There is often a need to "tag" tuples so that can be retrieved in the right way, in C-Linda this is usually done with strings[6]. A second output primitive is `eval`, which also has the more important function of launching new threads of computation:

```
eval("a",f(1));
```

has exactly the same result as the equivalent `out` statement, with the difference that the evaluation of the whole statement occurs in a newly spawned thread. This is of course particularly useful if computing the tuple element expressions are either computationally intensive or involve separate Linda communications (in this case that would be in the function `f`).

The primitives for reading tuples look very similar, with the exception that they may include formal variables that denote fields in the tuple whose value we want to retrieve:

[6]This is merely a convention though, tuples are allowed to have any shape.

```
in("search",?x);
```

would read the tuple written by the out statement above and bind x to 1. A field consisting of an l-value prefixed with a ? means that it can match any value (of the correct type), other fields have to be exact matches in order for the whole to match. in removes the tuple from the tuple space when the match succeeds, alternatively, rd functions exactly like in but leaves the tuple in place. Both will block if no matching tuple could be found, their predicate equivalents inp and rdp however instead will return a boolean indication whether the read operation succeeded or not[7].

We will look at two simple examples here, just to get a feel for Linda.

```
phil(int i) {
  for(;;) {
    think();
    in("room ticket");
    in("chopstick", i);
    in("chopstick", (i+1)%NUM);
    eat();
    out("chopstick", i);
    out("chopstick", (i+1)%NUM);
    out("room ticket");
  }
}

initialize() {
  for(int i = 0; i<NUM; i++) {
    out("chopstick", i);
    eval(phil(i));
    if(i!=0) out(''room ticket'');
  }
}
```

This is the well known "Dining Philosophers Problem" written in C-Linda (example copied from [13]). In this version, tuples are used to encode the "resources" that the philosophers compete for, and the philosophers themselves are a thread/process each. A philosopher will repeatedly try to get into the room (getting a "room ticket") and

[7]These have been added in later versions of C-Linda

grab two chopsticks, eat, and then return the resources. Because there are always fewer philosophers "in the room" than there are chopsticks (because there are not enough room tickets), always at least one philosopher will be able to eat so no deadlock can occur, and assuming the Linda implementation is fair to the extent that it can't starve philosophers, no livelock will occur either.

A second example is a simplistic parallel prime number finder, written in a result parallelism style [14] (result parallelism is where the processes involved build up a live data structure in the tuple space which becomes the result of the computation):

```
int is_prime(int me) {
  int limit = sqrt((double)me)+1;
  for(int i = 2; i < limit; ++i) {
    int ok;
    rd("primes", i, ?ok);
    if(ok && (me%i == 0)) return 0;
  };
  return 1;
}

int main() {
  int count = 0;
  for(int i = 2; i <= LIMIT; ++i) eval("primes", i, is_prime(i));
  for(int j = 2; j <= LIMIT; ++j) {
    int ok;
    rd("primes", j, ?ok);
    count += ok;
  };
  printf("%d primes\n", count);
}
```

The main function launches a process/thread for each integer to be tested for being a prime number or not, which will result in a tuple with the tag "primes" when finished. The function `is_prime()` computes the boolean that indicates whether a number is prime or not, during which it makes use of "primes" tuples of lower numbers that have already been computed (or not, in which case it will block until the tuple is available). This is an example of *live data structures*: an array of primes that evaluate towards a result while in the tuple space.

Unlike other concurrency models which are geared towards accelerating a specific class of parallel programs, Linda allows different styles of programming such as result parallelism, agenda parallelism (each process, or "worker", knows how to perform all tasks and processes whatever needs to be done), and specialist parallelism (each process knows how to do a particular task well and grabs only those tuples for processing) to be encoded. Tuples can have a great number of functions: they can be actual shared data structures, messages, tasks to be performed, locks, monitors, synchronisation points etc.

Besides C-Linda the Yale Linda group created a Linda implementation for FORTRAN[59]. Since then, lots of languages extended with Linda primitives have been created, for example Shared-ProLog [19], Astro-Gofer [24], ISETL-LINDA [25], Scheme-Linda[66], Linda for ProLog [65], SETL-E[33], and Eiffel[39]. A good overview of languages and systems based on the Linda tuple space mechanism is [54].

2.3 Sharing

This section will shed some light on part of the problems indicated in the introduction: *dependancies*, particularly those caused by *sharing*. Sharing is a central theme in programming languages, and many issues can be seen as originating from ways of dealing with sharing. Here we will focus on a particularly important form of sharing, *the sharing of mutable runtime values*. Models for this form of sharing form the greatest divide in programming languages, that between imperative and pure functional languages. As we shall see later, this is also where this thesis will make a contribution in the form of a new sharing model. But let us first look at the two main existing models.

2.3.1 Imperative sharing

In an imperative language the mutable values we are interested in are mainly dynamically allocated data structures, though the same holds for statically allocated memory (global variables and static data structures) as well. Access (the actual sharing) occurs by means of pointers (or, in the static case, through names)[8] (for an introduction to imperative sharing see also [32]).

The sharing model an imperative language gives you is the most liberal one possible: there are no limits on the ways sharing can be structured. More precisely, for any given

[8]Note that in more modern languages like Java, *all* datastructures are allocated dynamically, and subsequently nearly all access (bar class variables) happens via pointers.

value, we can say there exists a set of *observers*, i.e., those entities that have access to the value (a pointer to it). If the cardinality of this set is greater than one, we say the value is *shared*. In an imperative language, any one observer cannot determine this cardinality, and thus has no way of knowing whether a value is shared or not. We can say these observers are *transparent*, in the sense that changes to the value by one observer affect all the others, yet they don't need to (can't be) involved in the action as they are unknown.

This transparency gives imperative programming a lot of power and convenience: in a complex program with many interlinked data structures, an entity can refer to all data structures it needs access to without restriction, and should the situation require it, it can modify the data it has access to, all locally, and all without having to update or refer to any of the other observers. Mutation is a basic operation with predictable efficiency, without any need for organisation. Many algorithms, types of programs and operating system interfaces are expressed most naturally using these features, and many programmers find it easiest to model problems in this way.

But the downside is quite significant: exactly because a mutation is transparent to the other observers, and nothing is known about them, it is very hard to avoid consistency problems. As described in the introduction, sharing involving mutation can give rise to a dependency explosion. To be precise, when we talk about dependancies, we mean dependancies between code fragments that deal with sharing of (mutable) values, which is a subtly different metric from looking solely at the actual shared runtime values. From a set of n such code fragments each will have a (non-obvious) dependency on all others, without necessarily knowing what they are. If one were to add one extra code fragment this could add n extra dependancies, so we could say the dependency complexity is $O(n^2)$ worst case. Whether or not such complexity actually manifests itself depends greatly on the structure of an actual program. Cognitively, a programmer can keep an *overview* over a program, i.e., continue to understand all the intricacies of a program in detail, as long as the complexity of what he has to keep in consideration while focusing on a particular part of it stays below a certain limit. Once this limit is broken (and part of what this thesis tries to address is that the beyond linear complexity of dependancies caused by transparent sharing of mutable values helps greatly to push this limit), a programmer loses his overview over the code, at which point the rate of introducing bugs and other problems accelerates [26]. The situation is worse in the case of maintenance, where successful modification relies on an overview being reconstructed locally. The "quality" of this reconstruction determines how prone modifications are to bugs, alternatively it

determines if software can be extended properly or whether development is just plain stuck.

Some would argue that OO solves this, as well designed OO programs hide a lot of the mutation of an object behind methods. But employing data hiding does not guarantee that there no problems between the various accesses of the object, it just makes it easier to get it right, and sometimes easier to fix too. However data hiding is not a uniform medicine: certain algorithms or programs require complex interlinked access between objects, and one can structure that in one of two ways: within one object, but this essentially gets you all problems described above only then locally, or outside of lots of smaller objects that communicate in non trivial ways, which gets you the same problems as well, only then mainly in the glue code (see e.g. [35] for a discussion of problems caused by object aliasing / sharing).

2.3.2 Functional sharing

If imperative sharing was at one extreme of the spectrum of sharing models (allowing any form of sharing), pure functional languages are at the other extreme. I say "pure" because there is a class of functional languages that contain imperative features and that are traditionally called functional languages also, because they share the same background and ideas. We will not consider them here.

Functional languages could be said to not allow sharing at all. To be more precise, they don't allow mutation, specifically of course mutation of shared data structures. If data structures can't be mutated then you can never tell the difference between two observers looking at the same object, or two identical copies of an object, so for the purpose of the discussion (on the level of semantics) we can say functional languages don't use sharing at all. Obviously in implementation for many languages it's convenient to implement repeated occurrences of variables on the right hand side using sharing, but besides efficiency this is transparent to the programmer.

Not being able to tell the difference between values and their copies means an object doesn't have an identity, and this property that imperative languages rely so heavily upon is usually referred to as *object identity*. Object identity is the property of a object that if some entity transforms it, it is possible for a second entity to observe this change transparently (i.e. it doesn't have to be made aware explicitly of this transformation). This is not the case in pure functional languages: if two entities both have access to a value `x` then the first entity's transformation `new_x = transform(x)` is not going to

be picked up by other entities that have access to `x` . In imperative languages this is only natural: `x := transform(x)` instantly gives all entities that know about `x` access to the new version. The object referenced by `x` therefore has identity: unlike functional languages where everybody works with their private copy of an object, here everybody will always be looking at the same single object throughout all changes.

The lack of object identity imposes enormous restrictions on the programming model we know from imperative programming, and to get around these, a functional program then has to be structured in a significantly different way than it's imperative counterpart. Functions that mutate a data structure have to be rewritten such that instead they now construct a new data structure containing the new value, and all unmodified values from the input data structure[9], and return this new copy. Furthermore, all other functions that expected to work on this "modified" copy, will now need to be passed this new copy in sequence. This is a feature that percolates up: programs that work on data structures that need to be modified will contain glue code dealing with extra parameters and return values throughout. Similarly, if an extra value is needed in some function somewhere, all functions that call it need to be modified to supply that value (and their callers too). Also, modifying a value which is buried deep in some data structure effectively means traversing the whole data structure, and rebuilding it from scratch to put in place the modified value, whereas in an imperative language this could have been a simple $O(1)$ operation.

This seems like too much of a price to pay but in practice it turns out to be a great advantage. First of all, this single feature of functional languages does away with most of the dependency problems described in the section on imperative languages above in one go, which is not to be underestimated. Second, the clumsy ways of dealing with mutable data structures as described in the previous paragraph really are from an imperative perspective, a seasoned functional programmer will structure his whole program with a functional style in mind, which can greatly reduce the overhead in the manual "plumbing" of values around function calls, and wherever this plumbing is required, it serves the purpose of making dependancies explicit. Though it seems that on the whole the functional way should be worth it, it has a great many hurdles to overcome, as there are many more complex programs whose most natural description relies heavily on object identity, there are many problems in interfacing with other systems (particularly operating systems) in a convenient way, and last but not least many real world programs rely heavily on constant complexity of mutation operations[10].

[9] It will now be obvious why functional languages insist on pattern matching as a language feature.
[10] Needless to say, these are all the focus of the functional language research community.

2.4 Graphical languages

What a graphical language is, is intuitively clear, and most people will see the difference between a graphical language and a textual language immediately, yet it is less clear how to define the difference precisely as there is a continuous scale from textual to graphical with no clear border halfway. So instead let us look at some properties which make a language graphical[11] rather than textual:

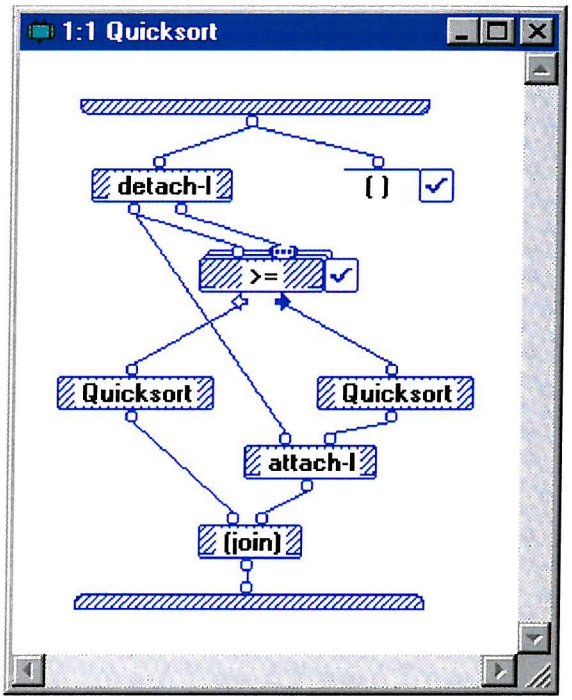
- A graphical language makes effective use of more than one dimension on a 2d surface. Textual languages are often no more than a one dimensional string of characters, spread out in 2d by using the linefeed character. A graphical language uses a diagram layout where the second dimension is at least as important.
- A graphical language uses graphical cues to denote relationships between program entities. In a textual language, almost all structure is denoted by nested brackets of some sort, i.e. {} [] () <> or matching keywords. A graphical language uses graphical relations such as *inclusion* (in a box, possibly hierarchical), *connected to* (a line between entities), and *composition* (on top of, to the side of).
- A graphical language will seek to reduce the need for identifiers to a minimum. A textual language uses identifiers for all relations in a program that aren't expressed by nesting and proximity. Nearly all graphical languages use a graphical construct of some sort to do away with the most common of all identifiers, the local variable. More global identifiers such as function names typically still require an identifier (or of course, an icon).

Graphical languages are slowly becoming more popular, though not as general purpose languages, but usually as simple, domain specific languages. To some, who superficially look at the idea of graphical languages, it seems obvious that with the added benefit of being able to use graphics in 2 dimensions, there is more opportunity for designing possible syntaxes and for making code easier to read, so it should be easy to make a graphical syntax that is superior to a textual one in every way. Yet somehow this isn't the case at all: there have been few graphical programming language designs targeted at general purpose programming, and just about all designs suffer from major impediments to make them useful as a replacement for large scale textual languages. In the history of graphical languages there have been 2 major families, and we will look at both to see what their problems are.

[11] The term "visual" is purposefully avoided as certain compiler vendors have blurred the meaning of that word as to mean "textual language with rich interface building facilities".

2.4.1 *Dataflow languages*

This is the biggest and most successful category of graphical languages, and there are many examples (e.g. Prograph [22], Labview [43]). In the simplest case, dataflow features 2 graphical elements, boxes and lines connecting them. A box represents an operation, i.e. the equivalent of an operator, a function call or sometimes even a control structure in conventional languages. Lines connect an output of one box to an input of another. An output can be seen as a return value of a function, whereas an input is an argument to a function. One can see a line as having the same function as a local variable in a textual language. This way, you can express just about any piece of code or algorithm. To make more complex programs, one can abstract over parts of this dataflow graph, and have a small subgraph that defines a function. This function can then be reused elsewhere by placing it in another graph. As an example, this is the recursive case of `qsort` in Prograph:



`attach-l` and `detach-l` cons and split the head and tail of a list, `(join)` appends a list and the special syntax around >= is a built-in list filter operation.

This system is very intuitive to understand and easy to work with, it has however one huge problem: as soon as dataflow graphs become more complex they look like incomprehensible spaghetti. Dataflow in its simplest form is a directed acyclic graph (a DAG), and there is no way to lay out all possible DAGs on a 2d plane without crossing

lines. Whereas DAG layout looks nice for the simple cases, the bigger a graph becomes and the more connections there are between boxes, the more lines will have to cross and the more messy things become. One trick programmers in languages such as ProGraph use is to cut up graphs to a great degree by making subgraphs into functions. This helps, but only slightly as connections that connect to boxes in the subgraph still have to connect to the outside of the function, and requires you to navigate a relatively large number of tiny functions to be able to review a relatively small amount of code.

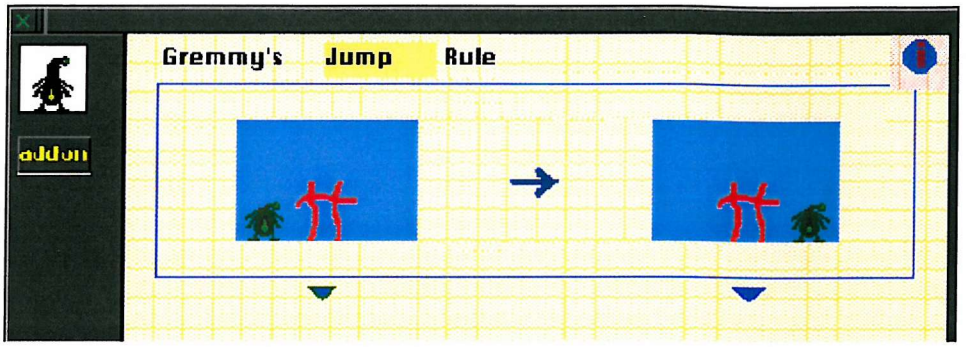
It seems that an apparent disadvantage of textual languages, having to come up with names for local variables and other items and having to spot the connections yourself, turns out to be an advantage compared to complex dataflow. Identifiers decouple complex structure, and have the side effect of giving things a sensible name you can remember. Though the fact that a complex overlapping dataflow graph is so hard to "read" may also be because we are so extremely used to reading textual code.

A last hindrance which dataflow shares with nearly all graphical language designs is its use of screen real estate. A couple of complex expressions which in a textual language would fill one or two lines of code can easily fill a whole screen. It will never be easy for a graphical language to compete with a textual one on the amount of code that one can fit on screen, but it can make up for this in easy of navigation of code.

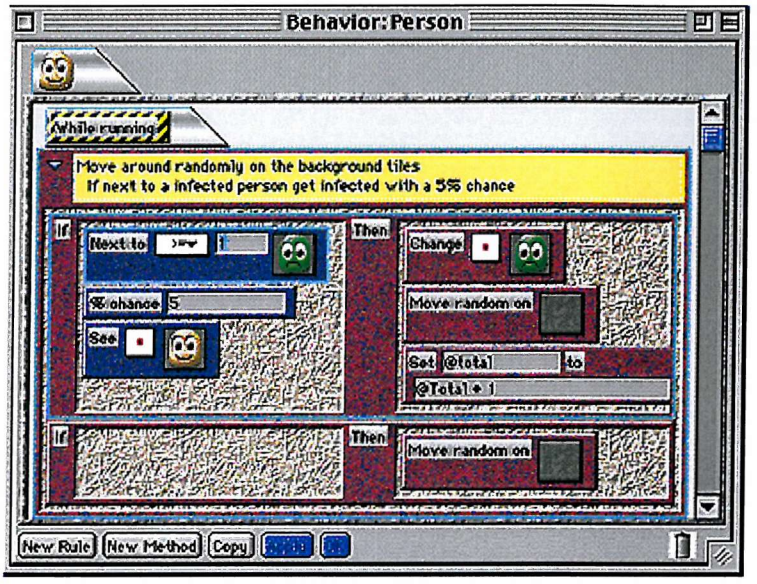
2.4.2 Icon / graph rewriting languages

There have been many graph rewriting languages (e.g. KidSim [63], Agentsheets [56], Chemtrains [9]), but most are less mature and less generally applicable than the dataflow family. Graph rewriting works in a similar way to tree rewriting described earlier, only now graphically: graphical structures left and right denote a rule which denotes how a graph can be changed should it match. The main graph, called the *world*, is the area to which these rules are applied until none are applicable anymore. The way the world (and the rules) can be structured differs greatly, this can be a fully featured graph (such as in Chemtrains and the graphical part of PROGRES [58]), or it can be a 2 dimensional grid of objects (such as in Cocoa[4] and Agentsheets, this is when we call a language an icon rewriting language).

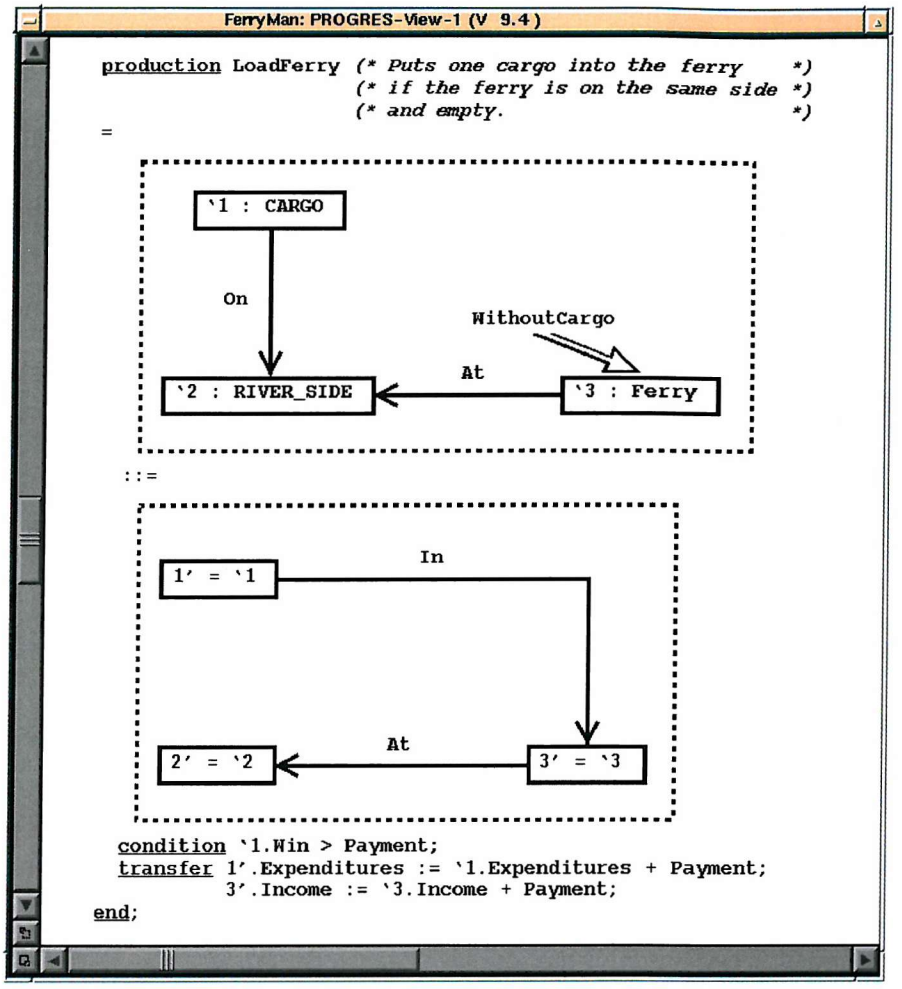
This example is from KidSim, which specifies a rewrite rule of "Gremmy" jumping over a fence in the world:



The following example is from Agentsheets, also a grid based language: it defines two rules for the Person icon, which determines how it behaves in the "world". Rewriting the world happens for all icons on the grid in pseudo-parallel.



The final example is from PROGRESS, a graph rewriting language with textual elements. Identity / placeholders carrying over from pattern to result are marked with numbers.



Graph rewriting languages have great potential for becoming usable graphical languages, as they can deal with complex data structures more easily than dataflow languages, and the before and after picture idea of a rule is very compelling one for programmers with less of a capacity to think in the abstract. Sadly, all current languages exhibit at least one of several flaws:

- Rewriting languages need placeholders in rules, and none of the current languages use particularly intuitive syntax for that. In particular when working with languages that are meant to be for non-programmers (Cocoa is made for kids) it is difficult to make intuitive the difference between concrete values and examples (those which are meant to be placeholders).
- Many of the languages are so targeted towards graphical manipulation that their world is something which is at all times completely visual on screen. This is fine for simulations and such where the program data corresponds closely to the on screen display, but is not suitable for processing more complex data structures.

- Graphs, though as mere data less complicated than their dataflow counterpart, still can be confusing. This why many of the simpler languages use a grid system, which, though obviously easy to layout and to understand, is very limiting.
- Nearly all of these languages have semantics that impede efficient evaluation strategies, because matching up situations in the world with a rule has a worst case of an exhaustive search (because there is no natural evaluation order in graphs). Their evaluation strategy is similar to normal order as presented in the chapter on tree rewriting, yet with a higher complexity because they deal with graphs and grids instead of trees.

Additionally these languages suffer from high screen real estate use like mentioned above.

Graph rewriting languages (and also dataflow languages) have failed to provide a syntax for mainstream programming, which is mainly due to the fact that their choices for graphical notation *don't scale up* to larger and more complex code, something which need not inherently be a problem of graphical languages.

Chapter 3

Aardappel, the language

This chapter will introduce the Aardappel language syntax and semantics. In the actual implementation of Aardappel the syntax is completely graphical, no textual implementation of the language exists. For simplicity of discussion however[1], this chapter introduces a textual syntax that is compatible with the graphical syntax.

3.1 Rewriting

The core of Aardappel is a tree rewriting language as introduced in the previous chapter, and specifically one with an eager evaluation strategy, specificity ordering, and dynamic typing. Recall that the benefits of eagerness are an efficient and predictable evaluation. Predictable will be especially valuable later on as we introduce state into the model. The price we pay is a loss in expressiveness, but as we shall see context sensitive expressive power is also gained when we look at Aardappel's rule selection order.

The core grammar of Aardappel is a refinement to that used in the previous chapter, made slightly more complicated because the grammar for *tree* has been split up into *pat* and *exp* to account for their different components:

[1] It would be inconvenient to litter this text with screenshots.

rule	=	*pat* [*patbag*] "=" *exp*
pat	=	*atom* "(" *cpats* ")" | *single* | *patbag*
exp	=	*atom* "(" *cexps* ")" | *single* | *expbag* | *expbag exp*
	|	*exp* "`where`" *rules* "`end`"
single	=	*atom* | *var* | *int*
cpats	=	*pat* ["," *cpats*]
cexps	=	*exp* ["," *cexps*]
patbag	=	"{" *pats* "}"
expbag	=	"{" *exps* "}"
pats	=	*pat* [*pats*]
exps	=	*exp* [*exps*]
rules	=	(*rule* | *pat*) [*rules*]
program	=	*expbag rules*

Note a couple of new elements in this grammar, most notably syntax for tree spaces (bags) and their corresponding Linda in/out operations, and rules local to an expression (`where` / `end`). Both of these will be discussed in the sections below.

3.1.1 Local rules

Local rules are rules local to an expression. This means that local rules get precedence over rules of a higher level (local rules can be nested to any degree, and obey lexical scoping rules): while reducing the expression the rules are local to, if a local rule is applicable then it will be used. Only if no local rules apply are other rules considered. So for example if we have:

```
angle90(vector(x1,y1),vector(x2,y2)) = test(x1*x2+y1*y2)
                                       where
                                         test(0) = true
                                         test(_) = false
                                       end
```

then `angle90` of two vectors would compute whether they are exactly 90 degrees apart or not, using 2 extra local `test` rules. Because `test` is local to the right hand side of the main rule, even if there are `test` rules at a global level they can't interfere with the evaluation of `angle90`.

A further consequence of lexical scoping is the possibility of *free variables*, which are supported. A free variable is a variable used in an expression part of a rule which

isn't available in the corresponding pattern part, but instead originates from a rule that encapsulates it. Since the rule with the free variable is always applied in a context of the outer rule, the free variable always has a valid value. Free variables combined with function values to form closures such as in functional languages would complicate matters, but these have been left out of the language in favour of making function values an idiom of the language, using tree rewriting itself to attain the same expressiveness (the next chapter discusses all this).

3.1.2 Normal forms

Part of the idea of local rules is of course shielding code from the rest of the world, and that's why rules explicitly allow normal forms to be defined. A normal form is defined in global rules as "has no rule applicable to it", but in local rules not having a rule applicable to a value means "try the global rules instead". This means it wouldn't be possible to make sure something is a normal form on a local level. Normal forms have always been implicit, but Aardappel allows you to define normal forms explicitly (both locally and globally) to improve the predictability of code. A normal form is defined by a rule without the expression part (just the pattern, see *rules* above). Without this, a set of local rules wouldn't be able to guarantee its behaviour independently of the global rules (modularity), as a global rule could apply to a local tree which was meant to be irreducible. For example:

```
a(x) = x
c(x) = b(a(x))
         where
           b(a(x)) = x+1
           a(x)
         end
```

Reduces `c(1)` to 2, not `b(1)`. If it weren't for the rules like `a(x)`, global rules could interfere with the evaluation of local ones.

3.1.3 Modules

Modules in Aardappel are simply a set of rules, and a program may consist of more than one module, which would be exactly the same as all rules of those modules placed in a single module. Additionally it is possibly to specify atoms as unique, meaning that if an identical atom occurs in another module these should be interpreted as different

atoms by textual substitution (they are renamed to be unique by the implementation): this effectively implements per module data-hiding.

3.1.4 Syntactic sugar

Local rules are part of the core of the Aardappel language, and can form the basis of a number of more convenient constructs which are defined as syntactic sugar on top of local rules. We can define an if-then-else expression as a local rule which matches true or not (note the _ is a "don't care" pattern):

$$\texttt{if } x \texttt{ then } y \texttt{ else } z \Longleftrightarrow \texttt{f}(x) \texttt{ where f(true)} = y;\ \texttt{f(\_)} = z \texttt{ end}$$

Similarly a local pattern match (`when`, to bind local names and extract values) can be defined as:

$$x \texttt{ when } p = e \Longleftrightarrow \texttt{f}(e) \texttt{ where f}(p) = x;\ \texttt{f}(y) = y \texttt{ end}$$

Unlike to a `let` or `where` expression in functional languages which can define any local value (not just a function), Aardappel only has local rules, and `when` is simply a short version of how to model a local binding using a local rule, for example:

```
twosquare(x) = y+y when y = x*x
```

is the same as writing:

```
twosquare(x) = f(x*x) where
                 f(y) = y+y
               end
```

Besides these two, we will use common syntactic sugar used for `cons` cells (square bracket lists) and tuples (multiple values included in parentheses).

3.2 Tree spaces

Rather than just evaluating one tree to normal form, Aardappel allows you to reduce a multiset of trees concurrently. Such a multiset of trees is called a tree space, and is a first class value in Aardappel. Besides being a bag data structure, a tree space is a more general version of a Linda tuple space, and inherits all its properties.

3.2.1 *Tree space evaluation*

Evaluation of trees in a tree space happens concurrently, and in the simplest case, the tree space is in normal form when all trees are in normal form. There is more to the evaluation of a tree space than mere parallel evaluation though: a tree, while evaluating, can consume other trees from the tree space, if they have reached normal form. For example, consider the rule:

```
a(x) { b(y) } = a(x+y)
```

The curly brackets in the pattern half of the rule denote a set of trees we want to take from the tree space (similar to multiple Linda `in` statements, only now atomic), and that have to be available for this rule to be applicable. So if we start out with a tree space `{ a(1) b(2) b(3) }`, trees `b(2)` and `b(3)` will be in normal form immediately, since there is no rule that applies to them. Next the rule above can do a 1 step reduction on the `a(1)` tree, and reduce the tree space to `{ a(3) b(3) }` (or `{ a(4) b(2) }`), and after a second reduction the final value is `{ a(6) }`.

So what happens if the only rule that is applicable for a tree needs another tree from the tree space which is not available? We say then that evaluation of the tree is *blocked*: no rule is applicable to it now, but might be in the future. This is basically a state which is neither actively computing nor in normal form. Whether such a tree can be called in normal form depends on the rest of the tree space: if all other trees in the tree space are either in normal form or also blocked, then it is impossible for the rule to ever be applicable, so the tree is then properly in normal form. This of course then holds for all blocked trees in a tree space at once, so a tree space with only blocked or normal form trees has reached a fixed point and is itself in normal form. It is called a *live lock* if evaluation of a tree space never reaches this fixed point[2].

Similar to the Linda `in` operation, Aardappel provides a Linda `out` operation at expression level. For example, the rule:

```
a(x) { b(y) } = { b(y-1) } a(x+1)
```

This reduces trees of the form `a(x)` by substracting `1` from the child of a `b` tree, then putting it back. Semantically, this corresponds with Linda's `eval` operation, as it will

[2]Which is usually but not necessarily a programming error: as Aardappel is a language that allows side effects in terms of input/output to the host operating system while a program is running, a program that is in a live lock can perform a useful task.

first output the tree(s) in the tree space for them to be evaluated concurrently, and then continue to evaluate itself. This is what the Linda people call *live data structures*: trees which reduce concurrently to their normal form. As soon as a tree has been put into the tree space it is on equal footing with the tree that spawned it there: it is not impossible for a tree to consume the tree it originates from.

Like any real concurrent system, order of execution between concurrently executing units is non deterministic. In the case of Aardappel this expresses itself in that we don't know anything about which order the different trees within a tree space will reduce to normal form (even if the evaluation is trivial), and consequently in which order `in` and `out` operations will happen, and in which order trees will be consumed. This means that it is possible to write programs that give different results on repeated executions, but in general Linda's automatic synchronisation by blocking on tree reads will mean that it is easy to create programs which give a deterministic result even though execution steps are non deterministic. The only guarantee made on concurrent evaluation globally is a minimal fairness requirement of *no starvation*: it takes a finite number of reduction steps to give every tree a turn in evaluation. Non deterministic steps such as taking a tree from many suitable ones from the tree space is irreversible: Aardappel does not perform any kind of backtracking.

As a very simple example of a tree space program, consider a program which searches a database (in parallel) for a number of words and gives the word with the highest frequency of occurrence as result. Such a program could be written in Aardappel as:

```
{ best(0,"none")
  search("apple")
  search("pear")
  search("banana") }

search(s) = isbest(score(s))
  where
    isbest(x) { best(y,t) } = if x>y then best(x,s) else best(y,t)
    score(s) = /* ... */
  end
```

Here `score` does the actual (sequential) search for one word and evaluates to the number of occurrences of that word. Note that we use a second rule to take the best tree (the current best result) from the tree space, as we obviously want to do this once

the score has been computed, otherwise the whole search would be nearly sequential. As shown in this example, a program always consists of a tree space and a set of rules[3]. The top level of a program is always a root tree space, because any rule may perform tree space operations, so any tree always has to have a tree space encapsulating it (not just a tree) for an evaluation to be meaningful.

3.2.2 Hierarchical tree spaces

As mentioned above, once a tree space reduces to normal form it is just another value which can be passed around and pattern matched upon. This also means tree spaces can be structured hierarchically to any arbitrary degree. Evaluation of nested tree spaces is still concurrent: as soon as a tree space is created (appears on the right hand side of a rule) the contents of that tree space is evaluated concurrently to the other trees reducing at a higher level. Normal form, however, is hierarchical: only once the tree space is in normal form can the tree that contains it continue to reduce and get to normal form itself, which in turn is necessary for its parent tree space to be in normal form etc.

Some important properties emerge from this which are important to note. A tree containing a tree space is essentially "busy" reducing while the child tree space is reducing. Reducing a tree is a sequential process, and reducing a child which is a tree space means the parent will continue to reduce itself as soon as the child is in normal form. The fact that this involves concurrent computation is something that is irrelevant to the reduction of the tree, it will never know anything of the concurrent computation or communication, as it can only look at the result, a multiset of normal form trees. This is because to put trees in or out of that tree space, the parent will need pass the tree space to another rule, which means the tree space already has to be in normal form (dictated by eager evaluation semantics). This means that while a tree space is busy reducing, it is effectively shielded from the rest of the world, it cannot communicate with its parent (it doesn't even know one exists), and its parent doesn't even have a notion of it being active. The only way to break through this tight encapsulation of concurrent environments is for the parent to explicitly manage the data in the tree space after the tree space has reached normal form, by adding / removing elements (which will cause the tree space to not be in normal form anymore, and reduction will start again).

[3]In the actual implementation this is a set of modules each of which contain a set of rules.

As an example, the expression `{ 1+2 3+4 blah({ 5+6 7+8 }) }` will have a maximum parallelism of 4 processes computing concurrently (imagine + taking a really long time for this to make more sense). The `blah` tree will be busy-waiting until both 5+6 and 7+8 have reduced to normal form, at which point the local tree space and itself will be in normal form, and the top level tree space can also reduce to normal form.

An example of a local tree space used to implement an explicitly parallel function:

```
qsort([]) = []
qsort([p|t]) = merge(sort(filter(t,qsortcompare(p))))
  where
    sort((a,z)) = { left(qsort(a)) right(qsort(z)) }
    merge({ left(x) right(y) }) = append(x,[p|y])
    left(x)
    right(x)
  end
apply(qsortcompare(x),y) = y<x
```

This version of `qsort` explicitly sorts both halves concurrently[4]. The local rule `sort` spawns the two halves, and once both have reduced to normal form, `merge` extracts the data from the tree space. The usage of `apply` is a standard idiom used in Aardappel code to emulate function values using tree rewriting, and is discussed in the next chapter. `append` was shown earlier, and `filter` could be defined as follows (both are standard library functions of course):

```
filter([],_) = ([],[])
filter([h|t],f) =
  if apply(f,h) then ([h|a],b) else (a,[h|b])
    when (a,b) = filter(t,f)
```

3.3 Rule selection order

Rule selection order may seem like minor detail, but subtle changes can have great effects on the dispatching, polymorphism and types of a language, as will become appearent in this section.

[4]Which is not as practical as it sounds, as concurrency overhead on the leaf calls would defeat the purpose, but it's a nice example nevertheless.

3.3.1 *Specificity ordering*

Aardappel's rule selection order is that of *specificity ordering*. This means that for all rules that potentially apply to a tree, the most specific one is the one that is actually used. What is more specific than something else is not easy to define universally, so Aardappel has its own definition of what is more specific. For the elements that can occur as part of a pattern we have the following total ordering:

$$variable < int < atom < bag < tree$$

A *variable*, *int* or *atom* is equally specific to an element of its own sort. A *tree* (and a *bag*) is ordered according to the specificity of the first child. If the first child is equally specific then the second child decides and so on. If two top level patterns are equally specific and also are identical with respect to all atom and integer values in the part of the expression that determines the specificity, then rules are said to clash, a situation which should be resolved manually[5]. A different way of saying this is that in Aardappel the source code order of rules has no influence on semantics, and all rules that violate that with respect to the above specificity ordering should be flagged as such.

The differences between this ordering strategy and a more conventional one like source code ordering only start to become appearent when we look at larger programs, specifically those consisting of multiple modules and/or are constructed by multiple people (either by working together or by reusing other people's code). When programming a large program using source code order, you basically have to perform specificity ordering yourself in those cases where you add a rule that is more specific than certain rules but less specific than certain others: you have to insert the rule at the right position in the program for it to be triggered by the right trees. In the case of modules, specificity ordering allows one to practically interleave all rules in all modules. The consequences are that one can hook into existing code without touching it, a feature deemed important by proponents of object oriented languages.

3.3.2 *Polymorphism*

Specificity ordering gives us a form of polymorphism which we can more easily classify if we look at the top level tree as a function or method, and its children as data types

[5]How an implementation conveys this to the programmers and what action is required is not specified.

(remember that in eager evaluation the top level of a tree that matches a rule is never in normal form, and its children are always in normal form). There are two topics that can shed further light on Aardappel's polymorphism: dispatching and compatibility of values.

The type of dispatching in a language tells us about the way a function is selected to operate on a particular object, should there be more than one choice. Here we are only interested in *dynamic dispatching*, i.e. the function to be applied to the object is determined dynamically, by looking at the "type" of an object. Object oriented languages give us two forms of dynamic dispatching, *virtual methods* and *multi methods*, for functions that dispatch dynamically only on the first argument or on all arguments respectively. Aardappel's dispatching (pattern matching governed by specificity ordering) is similar to multi methods, but with the subtle difference that pattern matching selects on structure rather than on type. This means that if a function pattern matches on the structure of a particular tree directly, there is no possibility to also work on "subtypes" as is possible in multimethods, because there is no way to create another (different) tree that is compatible with this function.

Closest to the notion of a "type" in Aardappel is compatibility of values, which is different from types of objects[6]. The set of all trees which share the same head atom doesn't correspond to a type as you have to pattern match on a tree to examine its structure, which means a particular rule may only apply to a subset of all trees with that head atom (because the structure of the children may differ). Value compatibility in Aardappel, then, is defined by the set of all trees matching with trees appearing at a certain position in a pattern with a certain root atom. In the function/object analogy: compatible values are the set of all objects that have a certain function implemented for them. For example:

```
f(a(b(x))) = x
f(c(x)) = x
```

Trees `a(b(1))` and `c(1)` are both compatible values for the first child of tree `f`, but `a(1)` isn't. Because a particular object can be a valid child of many trees that have rules applicable to them, it consequently can be part of many sets of compatible values (have

[6]This notion of type is useful to introduce to be able to compare Aardappel's polymorhism with other languages whose polymorphism is based entirely on types (as we will do in more depth in the next chapter), however Aardappel's they are not a language feature explicitly, merely a way of looking at it. Programs in Aardappel are always well typed with respect to value compatbility, so it has no value to the classical purpose of types to catch static or runtime programming errors.

many "types"). This corresponds to the typing system used in many object oriented languages called multiple inheritance, with the difference that in multiple inheritance all types have the subset relation for the objects part of the type towards any of their parents (i.e. the types are organized in a directed acyclic graph, or in the case of single inheritance, a tree), whereas in Aardappel no set of compatible values necessarily is a subset of any other[7].

Aardappel code uses this notion of compatibility throughout to attain inclusion polymorphism, often without planning. For example, the `qsort` example above is based on the fact that elements of the list it is sorting evaluate to a boolean value when placed in the context of a $<$ tree with another element: one could say, for us to be able to reuse the `qsort` code with a new datatype, it has to be compatible with $<$ (in multiple inheritance OO languages this would be a separate superclass / interface `Comparable` that a new class needs to inherit).

As an example we create a new datatype of natural numbers using the two constructors `succ` and `zero`, and define the $<$ operation on it to make it compatible with code that uses $<$:

```
zero < succ(x) = true
succ(x) < zero = false
succ(x) < succ(y) = x < y
```

now we can `qsort` a list of these values:

```
qsort([succ(succ(zero)), zero, succ(succ(succ(zero))), succ(zero)])
```

which will result in:

```
[zero, succ(zero), succ(succ(zero)), succ(succ(succ(zero)))]
```

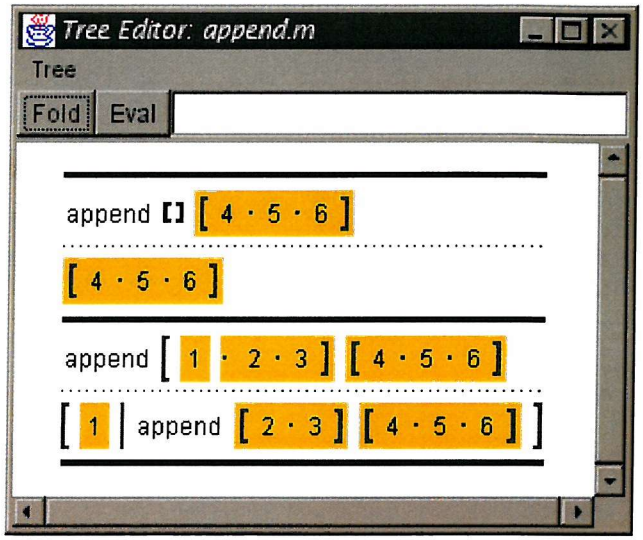
3.4 Graphical syntax

Aardappel was designed with a graphical syntax, mainly because its simple structure based on trees and rewriting is very suitable for a graphical representation.

[7] In OO languages each type is always a subtype of another, i.e. the set of interfaces it supports is an extension of (properly includes) what its supertype supports. In Aardappel, the "types" a certain value has is an arbitrary set, i.e. value `x` may have types `a`, `b` and `c`, whereas value `y` has types `a`, `c` and `e`. Neither is a complete subtype / supertype of the other, though they may be compatible with each other in a large number of situations (where something of type `a` or `c` is expected).

3.4.1 *Tree representation and editing*

Trees are laid out in 2d automatically as economical as possible, and any editing operations that the user performs on these trees cause a new layout: programmers don't ever organize the on screen position of program elements manually. Two trees separated by a dotted line constitute a rule, the pattern on top and the expression below. A list of rules are separated by thick horizontal lines:



Trees are selected by selecting their head atom. Dragging a tree from one place to another has the effect of cloning it and placing it on the destination position, alternatively when dragging values within one expression they can be shared to allow both copies on screen to be kept in sync, and updated simultaneously (shown graphically). This is essentially creating a DAG using a graphical tree syntax, and is similar to introducing a local identifier which is used twice (a `let` expression in functional languages or a local rule in Aardappel: `f(shared_expression) where f(x) = ... x ... x ...`).

3.4.2 *Examples as placeholders*

Rewriting is easily expressed graphically, but as to how to represent placeholders (variables) in rules there is no universally pleasing solution. The function of a placeholder is to allow the programmer to quickly spot equivalences in the pattern and expression, and in such a way that the placeholder is meaningful. Textual languages have always used identifiers, which completely decouple the locations of occurrences of the placeholders. In a graphical language however we want to avoid identifiers, as using them means we're not using the graphical expression power and no advantage over a textual language is

created. An alternative way to link multiple occurrences of placeholders is by using a graphical relation such as a line to link them. This is perfectly suited to a graphical language but in Aardappel we want to avoid them for the reasons mentioned in the last chapter w.r.t. dataflow languages: our neatly organized trees would become a graph.

In Aardappel placeholders are visualised as examples. Imagine this like programming with no abstract values at all: to create a pattern you create a fully concrete tree, and where you would normally have used a variable you now write any value you like, but one representative for the values you'd like to match, i.e. one that is a average case for your rule, or makes a good example. Now when you are constructing the expression part of the rule, whenever you need a placeholder value, you drag over that example value. By doing this, you mark the example value as an example rather than a concrete value, and the editor will show this by rendering it in a different way, in this case by giving it a special colour background. Once the rule is done, the placeholders are exactly those which have been used in the expression part as well, and marked accordingly. The programmer, upon inspecting the code, can determine whether something is merely an example or a concrete value. One thing that is not clear in this syntax is whether two identical values are necessarily representing the same placeholder. Here Aardappel relies on the programmer to choose sensible example values that do not clash with each other.

One thing here that is different from textual languages is that the editing operations are used by the system to infer the meaning of the code, you could almost say that an operation like dragging values from pattern to expression are part of the syntax of the language, as they determine code structure. Graphical languages, besides shifting structural complexity to 2d graphics, can also make use of the editing process to simplify syntax.

Other operations to enhance graphical programming result from the fact that every tree in the code can be viewed as a concrete value. For example when the programmer creates a new tree, the system can easily look up an example tree to fill in the children of the new tree with reasonable values. This gives an example of usage of the tree, and also makes the code instantly runnable as there are no missing bits. Another operation is folding by example: because every tree is fully concrete it can be evaluated on the spot and folded to whatever expression it evaluates to. This can be used to quickly check whether sub-expressions in your code do what you expect, rather than having to run the whole program to give variables a value.

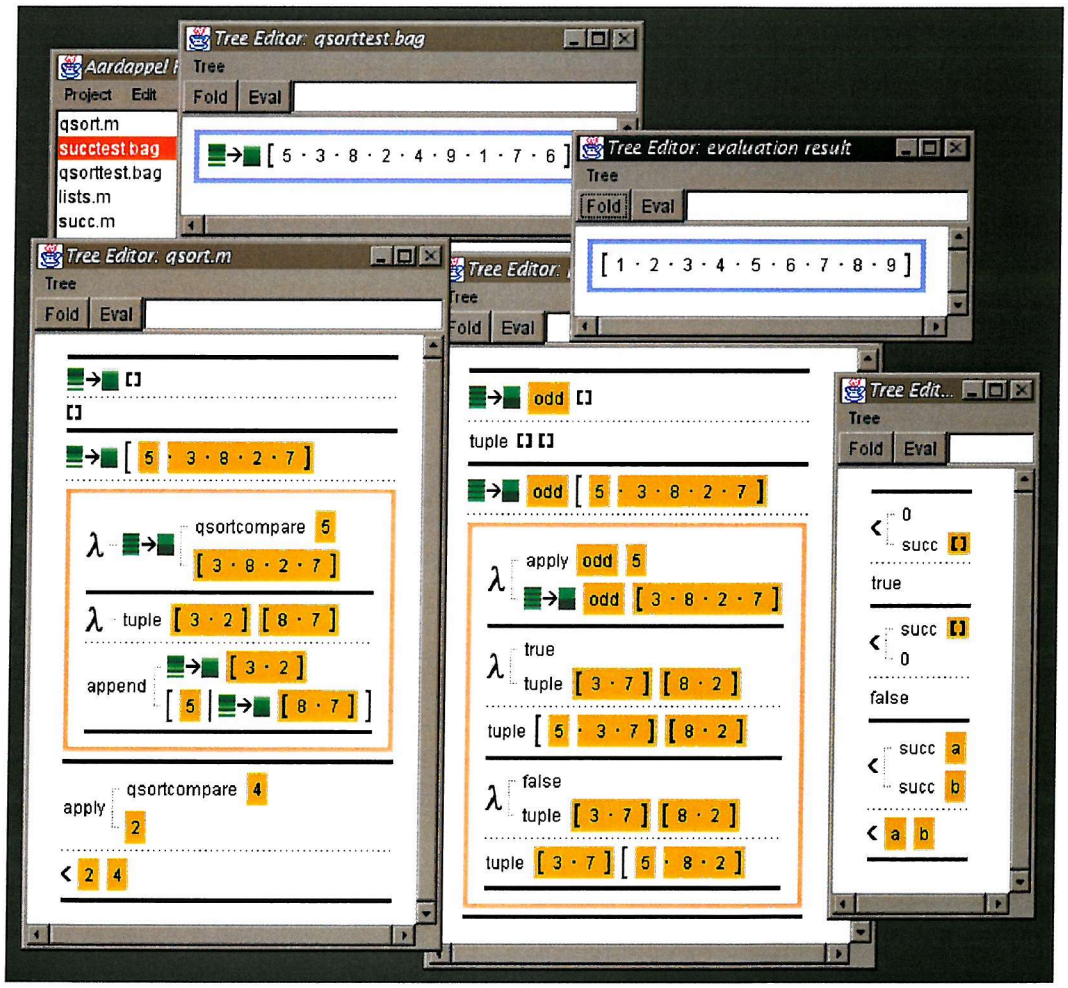
3.4.3 *Example: Qsort*

What follows is a complete example of a program in Aardappel, to demonstrate the graphical syntax. For convenient comparison, here is a textual version of what follows below in graphical form (this is the sequential qsort, the example above gave a concurrent version):

```
qsort([]) = []
qsort([p|t]) = append(qsort(a),[p|qsort(z)])
  when (a,z) = filter(qsortcompare(p),t)
apply(qsortcompare(x),y) = y<x
```

The `filter` function is the same as the one used above.

The quicksort code has been split up in a couple of modules, shown in separate windows below:



The Qsort algorithm is specified in the leftmost window. In this example, some identifiers are shown as icons, for example the `qsort` icon is the arrow going from a random set of lines to a sorted one, the icon for `filter` is a similar one going from interleaved lines to separated lines[8]. The first rule in `qsort.m` rewrites `qsort([])` to `[]` (nil). The second rule is the recursive case: the trees marked with orange[9] colour background are the example placeholders. The pattern thus reads `qsort([pivot|rest])` where `pivot` is the example value 5 and `rest` is `[3,8,2,7]`[10]. As is obvious already, the example values have been strategically chosen as typical examples and to be in tune with each other: 5 splits the list perfectly in two, and after splitting the two lists are unsorted.

The body of the recursive case introduces a local rule, marked by the pink-orange thick-line box. Within this box, the top expression is the resulting expression, after it has been reduced to normal form by the union of the rules below it and the global rules. As said before, the raw local rules construct is being used in all current code instead of if-then, `where` and `when` constructs as a visual syntactic sugar hasn't been implemented for them yet, and as you can see isn't really necessary. The local rule in qsort is the simplest one used in this code, it emulates a when-expression (or pattern matching let-expression). The lambda symbol is the default atom used as a new anonymous identifier but has no special semantics within the language, it could be replaced by any atom and retain its meaning. The rule evaluates thus: first it performs a `filter` with as arguments a new tree called `qsortcompare` containing the pivot, and the rest of the list. The result of this (a normal form) will then become part of the lambda tree, which will trigger the evaluation of the rule below it. The rule for lambda pattern matches on the result returned by `filter`, a tuple with two lists. The end result of the whole is then produced by quicksorting the individual two lists, and appending the results together (after consing the pivot on top of the second list).

A couple of things should be noted here. First of all, the pivot is used within the lambda rule as a free variable. Free variables are allowed for convenience, as they are equivalent to adding extra arguments to the local rules. Second is the use of the `qsortcompare` tree. Aardappel has no special feature to support function values and higher order

[8]This is a feature provided by the implementation: it allows you to associate a picture of your choice with any atom.

[9]Or medium-grey if the screenshot is not printed in colour.

[10]The editor decides dynamically how to render trees in a way to use screen real-estate most optimally: trees whose children are flat (i.e. single values or simple trees) and that can be displayed unambiguously as a flat list instead of a tree will be displayed linearly. Similarly, the `cons` tree, being one of the most frequently used trees, has a variety of special syntaxes to make it easier to read and use less screen real estate (mostly similar to ProLog list syntax). As you can see, this causes almost all trees in the first window to be rendered in flat mode, except for `filter`, `append` and `apply`.

functions, as the power of its specificity ordering allows it to get the same functionality from a programming idiom rather than a semantic feature (more on this in chapter 4). A function that requires a function value as argument (such as `filter`) requires that you supply it with any value such that when it is placed in the context of an `apply` tree it will evaluate to whatever the function value is supposed to do. In most cases this value is a new tree, with as children any values it may need (those that would be free variables in a functional language). Again, the atom `apply` has no special semantic significance, it is merely a convention used by all higher order functions and their callers. The next rule (and the definition for `filter` in the next window) defines this behaviour: `apply` of `qsortcompare` and an actual argument (the one passed from the higher order function) is defined a simple comparison of the two values involved.

The definition of `filter` is similar: the base case matches an empty list and returns a tuple of empty lists. The example function value we are using is `odd`, which is merely an example of a simple use of `filter`. the recursive case takes the head and tail of the input lists, and again uses a local rule for the body. This local rule is slightly more complicated though, as it is used as an if-then-else and a pattern match at the same time: it places the function value and the head in the context of `apply` (which will then use the appropriate rule as implemented by the caller to produce a boolean value), and makes a recursive call to itself with the tail list. Two different local rules are then used depending on wether the result of the former is true or false, and the end result is a new tuple with the head placed in either the first or the second list, depending.

These two functions together define what is needed for `qsort`. The top window is a sample call for `qsort`: it is placed in a top level bag, but in this case we use just one tree. Selecting "evaluate" on this expression will result in a new window, the one to its right.

The last window on the screenshot is there to demonstrate yet another way the power of specificity ordering can be used (to provide inclusion polymorphism, as described in the previous section). The < atom used in the definition of `qsort` (or `apply` rather) is just another tree, meaning that any tree that when placed in the context of < can evaluate to a boolean value can be used by `qsort`. To demonstrate this, the rightmost window uses a new datatype (natural numbers, defined by `0` and `succ(n)`), and defines < for it, which allows lists of them to be quicksorted (not shown).

Chapter 4

Design discussion

This chapter looks at the language elements introduced in the previous chapter, and discusses their merit. The first two sections emphasise the gains obtained from Aardappel's version of tree rewriting and its Linda based concurrency. The section on sharing focuses on Aardappel's sharing model, a model that emerges from the tree space design, and something that is deemed to be the biggest strength of the Aardappel design overall. Finally, the last section gives an evaluation of what Aardappel's graphical syntax is worth.

4.1 Tree Rewriting

Aardappel makes some subtle changes to the base semantics of tree rewriting, but already the base computational model is as good as any. It combines great semantic simplicity (almost competitive with lambda calculus in this area) with expressive power, particularly its intrinsic suitability for transforming complex data structures. It treats all dynamically constructed elements in the language as one thing: trees (data structures, function calls, and in Aardappel: processes, tuples, messages, function values etc.), and especially in its incarnation as Aardappel it is able to express almost any algorithm as easily as much more complex languages can.

4.1.1 Multiple argument and multiple type inclusion polymorphism

The previous chapter outlined Aardappel's inclusion polymorphism, resulting from its specificity ordering in rule pattern matching. Inclusion polymorphism is a form of polymorphism most well known from object oriented languages, and it is compared to this

form of inclusion polymorphism that Aardappel has the apparent shortcoming of not being able to "pattern match on subclasses" (this is not even a meaningful concept for Aardappel). We will look at this issue in a more subtle way now.

In object oriented languages, methods can operate on objects that are a subclass of the class the method was written for, by virtue of the fact that every subclass is always structurally compatible with its superclass. Type compatibility in object oriented languages is strongly dependant on its inheritance feature, which creates new classes as extensions of the internal structure of a superclass. The fact that methods can operate on a subclass object is merely because it has the same structure so a method can treat a subclass object as if it was of its own class. This mechanism works well in practice but also has some problems. The subclass relation is there to ensure one very useful property: that for the clients of an object, the object's interface will be compatible with that of its superclass. When interfaces are compatible, it seems odd that the subclass is forced to have all the instance variables of the superclass: there is no reason why the implementation of the subclass couldn't be completely different. In just about all object oriented languages this situation means that a subclass object carries around a copy of the unused superclass structure merely for compatibility reasons. Furthermore, the writer of the subclass is not restricted as to which methods are rewritten specifically for the subclass, as the methods that are not redefined can simply use the superclass versions thanks to structural compatibility. This means that in a typical execution subclass and superclass methods intermingle, and superclass methods treat the object as if it was a superclass object, with no respect for the new instance variables and the new behaviour added by the new methods. If the new state and/or the old state of the object has some invariants to uphold, it is not difficult to create code that unwillingly makes a mess of things. Add to this the fact that it matters which methods actually modify/examine the state (you could call these the *core* methods) and which call just them, and that implementations of either may change (even by different people). If it weren't for the fact that inheriting state and the methods that work on it can be very useful, it would be best to do away with it altogether and do subclassing only by interface (as is possible in languages like Sather[52] and Java[30]). Making redefinition safer would require isolating core methods, and require they be replaced as a unit, or not at all.

In Aardappel none of this is an issue, as it is not possible to define trees which are structural extensions of another tree. If you create a new tree (i.e. with a new head atom and/or a different number of children), it will have no rules of existing trees applicable to it and all have to be created from scratch. Of course this means it has

none of the aforementioned problems with inheritance, but there is another logical reason why trees are better left structurally unrelated: trees matched by the main pattern have no object identity[1], they are fully consumed then recreated from scratch again (or not), as necessary. A "superclass method" would have no concept of modifying just a part of a tree, as trees are never modified, they are immutable data structures. Extended data that a rule doesn't know about wouldn't be able to be carried across, as a rule has no idea which two trees are meant to be the "same" (i.e. pattern matching loses the identity of the trees involved). The trade-off is of course that a lot of extra glue code needs to be written when there is a situation where what you really want to do is inherit instance variables (i.e. you want to change just a tiny bit of behaviour for a type of tree which has some complex code defined on it), but at least this extra code serves a useful purpose of making the relation explicit (you have to code the relation between the old and the new structure yourself). For example, to create a new datatype `cpoint` that not only reuses the representation but also code of a previously defined `point`, I need to define a rule such as:

```
rot(cpoint(point(x,y),c),angle) = cpoint(rot(point(y,x),angle),c)
```

(to remain polymorphically compatible with the operations defined on it, such as `rot`).

Leaving inheritance of state aside, what inclusion polymorphism in object oriented languages is all about is compatibility of interfaces: informally, that an object can respond to a certain set of methods. However in object oriented languages there is the requirement that these interfaces be organized hierarchically using the subtype relation, either with just one supertype (single inheritance) or any number of them (multiple inheritance). This means that in OO languages it is not possible for two objects to have a common set of methods they support, as well as both having a set the other doesn't support: class libraries have to be rigorously designed such that these cases are not necessary, and if a superclass is not available for modification, adding functionality to it (that the subclass does or does not require) can be problematic.

Aardappel's notion of compatibility of values (types/interfaces) is not restricted to an organisation of any kind. Implementing a new tree requires rules to be implemented that need to access that tree's inner structure: these would be similar to "core methods", and also gives a tree one or many "interfaces" (makes it compatible with other trees). Only exactly those rules that are needed to provide an interface for the situations the

[1]We emphasise the main pattern, as below we'll see that tree spaces *do* provide us with object identity.

tree needs to be compatible with need to be specified. Similarly, code that can use trees polymorphically through inclusion polymorphism can use existing interfaces to access trees and requires any trees that want to work with it to implement that "interface", or it can create a new, specific interface. The downside is that both the "definition" and the use of an interface is implicit, nowhere but in the rules is it apparent what trees are compatible with. From a tree rewriting (rather than an object oriented) perspective this comes very natural: a tree as an "interface" is a contract: clients only need to know they can wrap objects in a certain tree to get a certain effect, and implementors know they have to write rules for their own trees to make sense in the context where compatibility with a certain set of values is required.

As an example take a hashtable. In Aardappel, that hashtable can simply wrap each object it gets handed in a `hash` tree, and any tree that is needed to be used as a key in a hashtable needs to define a rule that rewrites it in the context of a `hash` tree. For example, the hashtable's `get` operation could look like:

```
get(ht,key) = ...h... when h = hash(key)
```

If I now want to my datatype point as a key in a hashtable, I write:

```
hash(point(x,y)) = x*y
```

If a certain tree has a rule defined for it in the context of a `hash` tree, we could say that tree is compatible with (amongst possibly others) the `hash` interface. In an object oriented language (assuming it supports multiple inheritance or multiple interfaces, otherwise there may be further complications if the tree already supports another interface), we have to inherit from a class/interface such as `Hashable`, and implement a `hash` method accordingly. But if you want to use the hashtable with a class (say A) that you haven't written yourself you are stuck, since you can't make A inherit `Hashable` (apart from retro-editing classes being bad style, you may not even have the source). Subclassing A isn't going to help either, because then clients of your code that already use A objects will then be stuck. In Aardappel none of this is ever a problem, regardless of whether the objects you want to define the operations on are language built-ins, objects from a class library, or completely new ones, and regardless of whether you are the implementor of the class or not, and which existing code you need to support. This allows for reuse and extension *without planning*, something which current object oriented languages don't support very well.

To summarize this difference in inclusion polymorphism, object oriented languages are good at extending a program with new types, but extending it with new methods that

work on existing types is complicated. In a modern functional language it is easy to extend a program with new functions to work on existing types, but adding new types that work with existing functions is difficult. In Aardappel both are easy to do, with the only difficulty being that you have to implement all rules that access structure from scratch, catering for the contexts the new tree can be used in[2].

4.1.2 Local rules, free variables and function values

Unlike functional languages, a tree rewriting language (and Aardappel in particular) is quite minimalistic in the sense that it doesn't have (and doesn't need) control structures such as conditionals or let expressions. To take a tree expression and examine it further as to whether it satisfies a certain predicate (as in a conditional) or share its value (as in a let expression) we simply pass it on to another rule to be taken apart further. This style of "concatenating" rules to accomplish more complex goals is very common to tree rewriting, and causes many helper rules to be created. This means that the feature of local rules is more essential than in other languages, as all these helper rules greatly clutter the global namespace, especially because they are only needed by one rule.

Even if local rules are essential to make programming in a tree rewriting language not too cumbersome, their mapping onto global rules is so trivial it makes sense to think of them as syntactic sugar, to the end of keeping the core semantics of the language simple (this is also done in the formal semantics in the next chapter). Any set of local rules can be made global by textually replacing the head atoms of each pattern and their occurrences in the expressions within the scope of these rules by globally unique atoms, and to add any free variables occurring within the right hand sides of those rules as extra children to those patterns. For example, a rule such as:

```
a(x) = b(1) where b(y) = x+y
```

Can be replaced by these two rules:

```
a(x) = unique_b(1,x)
unique_b(y,x) = x+y
```

Originally Aardappel had proper closures built in to the language which makes free variables a lot more complex than the mere convenience of saving an extra argument they are now. Closures were dropped from Aardappel because tree rewriting already

[2]This will be less of a burden if programs are constructed in a style to access objects on type rather than structure, which will also greatly increase the use of polymorphism.

makes them redundant (they are almost as easy expressed as a language idiom). It is interesting to compare the original function value as a language feature, and as an idiom.

Aardappel allowed atoms to function as *closures*, similar to function values in functional language. Such a closure consists of an atom (a tree without any children), and free variables used in the rules potentially applicable to this tree, if any. A closure was denoted by the familiar `atom/arity` syntax from ProLog[3]. For example:

```
makeadd(x) = f/1 where f(y) = x+y
twice(f,a) = f(f(a))
```

This is a clear example how closures and free variables cooperate: because we can create closures using local rules which are then passed on anywhere, the value of `x` is stored in the closure created by `f/1`. For example if we evaluate `twice(makeadd(2),10)`, when `twice` applies the closure to a child in `f(a)`, the value of `x` is made available again so the local rule can be used correctly as if it were evaluated in the context of `makeadd`. This is the same technique used in functional languages to implement function values, only now made suitable for tree rewriting.

Closures were clearly the most complicated feature of the tree rewriting part of Aardappel, and they don't fit naturally (they feel a bit out of place, compared with functional languages). Emulating function values using vanilla tree rewriting however gives code that feels very natural and suits tree rewriting very nicely. A function value in Aardappel is just another tree. Triggering code when it is "applied" to arguments is done by making use of specificity ordering. Let's look at a classic function that needs a function value, `map`, and pass it a function that uses free variables[4]:

```
map([],_) = []
map([h|t],f) = [f(h)|map(t,f)]

scale(l,n) = map(l,f) where f(x) = x*n
```

[3]This is needed to indicate what kind of tree "applying" the children will result in, and thus what set of rules is potentially applicable (needed to determine which free variables are to be part of the closure).

[4]It should be obvious that only when free variables and function values can be used together, things get interesting: in semantics because it requires closures (and on the implementation side, garbage collection). For the programmer, the combination of the two opens up very expressive programming styles. Without each other, they aren't nearly as useful: Pascal has free variables, and C has function values, but they are not used nearly as much as in functional languages.

So `scale([1,2,3],2)` would evaluate to `[2,4,6]`. Rewriting this in Aardappel without making use of language level closures is easy:

```
map([],_) = []
map([h|t],f) = [apply(f,h)|map(t,f)]

scale(l,n) = map(l,scale_item(n))
apply(scale_item(n),x) = x*n
```

This is an idiom, where we create closures ourselves as any tree we like (any free variable we can conveniently pass on as children), and a certain atom used by both rules that use function values, and those that define them. Since there is no reason not to use the same atom all the time, we use the suitably named `apply`. Creating a function value is just creating any value you like (as long as it is unique compared to other function values), and making sure it reduces to something in the context of the atom that is used to trigger evaluation. This is a convention that makes use of the strengths of tree rewriting, and as such comes very natural. It is hardly more verbose than the language built-in version.

By far the biggest disadvantage of this idiom is that it requires the introduction of a globally unique name (here `scale_item`), because our regular trick to reduce global names (local rules) can't work here. The idiom relies on Aardappel's specificity ordering to make our function value part of a huge global function called `apply`[5]. It is very easy though to introduce additional syntactic sugaring to alleviate this, and allow programmers to simply use something like `lambda(x) = x*n` instead.

4.1.3 Dynamic typing

As mentioned before, dynamic typing is hardly a design choice when it comes to tree rewriting but more of a given. The advantages of dynamic typing are well known, and include greater expressivity, easier rapid prototyping, less clutter in program texts, and unrestricted parametric polymorphism. The disadvantage is that feedback on program errors, especially of the "illegal argument" variant, shifts from compile time to run time. Worse yet, in a tree rewriting language there is no concept of illegal argument at all, since constructing a tree that doesn't work with a certain rule simply means it is in normal form. Often this should be relatively easy to debug though, as a program that

[5]Because map never actually accesses the function value, the function value name can be made local to a module.

fails because somewhere along the line an "illegal argument" was supplied to a rule will result in a tree, where one of the leaves will be the culprit value. For example if I wrongly write the expression `a(c(1))` where I meant to write `a(b(1))` because that is what my rule was meant to work on, the result of my program will be `a(c(1))` encapsulated in all trees that also didn't evaluate, because they expect to work on whatever the a rule results in instead.

This may sound inconvenient but it is simply a different way of working. There is a great deal a language environment can do to improve the situation, the most important one being to provide type inference and analysis in the implementation. In statically typed languages, type inference is used as part of the language and the static type system, but there is no reason why it can't be used in a dynamically typed language as well, as besides giving warnings/guidance at compile time it can be essential for program optimisation. Our "illegal argument" can result in a warning fairly accurately, because of the roles trees tend to take. If an atom only occurs as the head tree of patterns, you can call it a function, and similarly if it only occurs as a child in patterns, it takes the role of data structure (because in eager evaluation by definition it has to be in normal form). If it occurs in both positions, less can be said about it but this doesn't occur a lot in eager evaluation (it makes more sense in normal order evaluation where this makes for a very powerful programming style). For these functions it is possible in many situations to see statically when a child of a tree can cause a tree to not match any and give a warning. Similarly using this information it is possible for an environment to automate the process of finding the culprit from a large expression when a program evaluation returns an unexpected result (it can simply highlight the trees that can be classified as "function" and have no function trees below them, see section 6.3).

4.2 Tree spaces

In chapter 2 the Linda tuple space model was introduced, which was shown to marry a very simple semantics with an impressive list of features and good properties, all of which carry over to Aardappel. In this section we will see how Aardappel's version of Linda is even simpler, and provides even more functionality. The greatest feature of tree spaces is the sharing model that emerges from it which is discussed separately in the next section.

4.2.1 *Aardappel vs C-Linda*

C (together with Fortran) is the first language for which Linda was originally implemented. Compared to Aardappel however, the marriage is a rough one: C-Linda differentiates between tuples and processes, and you have to manually specify whether you want the tuple to be evaluated in its own thread or not (`eval` or `out`), whereas in Aardappel there are just trees, which are always evaluated as a separate thread (similar to `eval`). Alternatively, you could say that in Aardappel all data structures are live data structures. Besides processes, the concept of a data structure is completely different in C and Linda (structs and tuples respectively): they live in separate worlds and translation is needed between them. Again in Aardappel, there are just trees. Selection of tuples in C-Linda is done by pattern matching, a feature not present in C. In Aardappel pattern matching of tuples is no different from the pattern matching found in the sequential part of the language.

There are two features where Aardappel extends Linda and thus could be argued to be more complex. The first is taking multiple trees from the tuple space at once: instead of specifying one tree you specify a set of them. The reasons for having this feature is to simplify programming, and more importantly, reduce the possibilities of deadlock: Linda systems not having this feature allow processes to interleave their `in` statements which may result in two processes only getting half of the resources they ask for and be blocked forever. Conceptually it is not a great burden, the only place where it may complicate matters is in the implementation (it will be harder to make efficient). The second is hierarchical tree spaces. As we will see next, this doesn't complicate things at all, as each single tree space still has the same semantics as a one tree space model.

4.2.2 *Hierarchical tree spaces*

Many structuring mechanisms in programming languages (i.e. hierarchical ones like those based on lexical scoping) assume that a nested (local) entity can always access its parent (global) entity. This puts a burden on top levels of a more complicated hierarchy as everything there is globally accessible, in a sense a local unit can never be seen as an isolated environment as it stacks upon others.

A nested tree space does exactly the opposite: it may be nested but the inside of a tree space has no relation to any parents of the tree space, the semantics of a tree space are exactly the same regardless of whether that tree space is the only one in the program, or whether it is part of a hierarchical set of tree spaces. A local tree space thus has no

access to its parents and similarly a top level tree space is not any more accessible than any other tree spaces in the program, because it is not being accessed from its child tree spaces.

This reverse scoping model turns the world upside down, and allows for truly isolated environments for concurrent computation and state. A tree space becomes the "unit" for concurrent communication, and a set of concurrent components (trees) can be put together in such a tree space for a specific goal (a specific client / server ensemble setup), and would be completely shielded from the rest of a more complex application. Besides using concurrency for structuring reasons, we can use tree spaces to build explicitly parallel functions, whose parallel implementation is completely transparent to the entities that use it. To summarise, hierarchical tree spaces give us *data-hiding for concurrency*: it can shield sets of concurrently communicating entities from each other much like normal data-hiding shields data structure access.

The only communication that can occur between two tree spaces, whether they are nested or simply siblings, is through the explicit transfer of data done by a tree governing at least one of them. This is something the tree spaces don't have any influence on. The fact that a parent tree of a tree space accesses it by pattern matching on the normal form of a tree space (i.e., the tree space as a mere data structure) means that even the parent tree can never interfere with the concurrent computation of a tree space.

4.2.3 Tree spaces as data structures

The main purpose for the existence of tree spaces is to serve as a dynamic environment to host concurrent communication and state, but since they can be created dynamically much like any value, they can also be viewed as data structures in their own right. Here we look at some of the properties of a tree space compared to the most well known data structure, the object oriented "object". A tree space is different from an object and both are structured in different ways, but it makes sense to stress the differences, given that both can be used as environment for code to refer to.

- Members of a tree space are available dynamically rather than statically. If, for whatever system the tree space implements, it makes sense that a certain item is not available upon creation, or similarly an item has no meaning anymore after the tree space reaches a certain state, these can simply be added/removed as the system dictates it. A normal object doesn't have this kind of support for evolving a data structure over its lifetime, and as a consequence it can be harder to maintain

invariants that have to hold at all times during the lifetime of an object, or track dependencies between valid states of members. The contents of a tree space can evolve over time to a completely new set of members.

- Tree space elements can be safely modified by multiple processes (actively evaluating trees in the tree space) at once, if there are ever two processes trying to access the same tree one will automatically block. Depending on how a tree space is split up into individual trees, these trees can be modified concurrently, whereas an object almost always requires a lock on the full object.
- Tree space elements naturally extend to multiple items of the same kind[6], you can build whole streams of a certain tree and have them processed in exactly the same way as a single tree. Such a stream has a similar functionality as an asynchronous unbounded buffer between producers and consumers of certain tree types, and no planning is needed to use this functionality.
- An element of a tree space is a non mutable value accessed by pattern matching, not a mutable field (as in an object).

4.3 Sharing model

Recall from the section on sharing models that there exist essentially two extremes in the sharing of mutable values: the imperative way (everything is possible) and the functional way (nothing is possible). This section will introduce Aardappel's sharing model which in some way hovers between these two. From the previous section it will have been obvious that the semantics governing tree spaces (specifically hierarchical ones) is quite special, and this section is all about making the sharing model that emerges from it explicit and to evaluate its merit.

4.3.1 State

Aardappel allows state, i.e. programs can be implemented in a non referentially transparent way. I use the more general term state here on purpose, as saying that Aardappel allows for shared mutable data structures is only partially correct, as we'll see in the next sections. State is introduced into Aardappel's semantics through the Linda in / out operations, which allow a tree to reduce making use of a second tree in the same tree space: this second tree is then the state which affects the evaluation of the tree. For example, consider the rule:

[6]Where "same" here refers to the same type as discussed in the section on polymorphism; effectively the same to whichever pattern is consuming these kinds of trees.

```
a(x) { b(y) } = { b(y+1) } a(x+y)
```

In the tree space `{ a(1) b(1) }`, `b(1)` can be considered a piece of state in this tree space, read and updated by any tree that uses the above rule and maybe others. It has object identity because all trees that access it refer to the same tree, and having copies of it would matter for program semantics. `a(1)` and similar trees like it are non referentially transparent in evaluation, i.e. two occurrences of the exact same tree may evaluate to completely different numbers.

There are a couple of things that make this kind of state different from the imperative model of updating memory locations. Most important of all, access to this state is completely local to the current tree space, and is subject to the same airtight shielding that governs hierarchical tree space evaluation. For trees to share state, they must live in the same tree space together. Second, all access to state is regulated by Linda's concurrency mechanisms, meaning synchronisation and blocking if multiple entities try to access the same bit of state at once is a natural feature. This makes access to state loosely coupled and robust: *you don't have immediate access to state, but you may request it.* To further aid this property, accessing a bit of state involves pattern matching, meaning that the structure of value has to be traversed from scratch. This ensures that code can't make assumptions which are not explicit about the structure of the state (often invalidated by updates of other processes), which avoids the classic fragility that complex imperative sharing often triggers.

4.3.2 Linearity

If the previous section explained how Aardappel's sharing model allows for state just like imperative models, this section shows why it is a far cry from the imperative "everything is possible" model.

Linearity is the property that all runtime data structures are referenced exactly once, i.e. have exactly one parent[6][7]. To put it in a different way, objects can at most form a tree structure, never a graph (even a mere DAG). For linear values to be used more than once, they have to cloned. Linearity in a referentially transparent language can be seen as a mere implementation issue (it determines whether in place update can happen without violating referential transparency, or whether cloning can be replaced by sharing etc.), but in a language with true state it becomes a crucial property of the semantics.

The previous section showed that state is available only locally, so what is the semantics of this state seen from a global level, looking top down along the whole hierarchy of trees and tree spaces? Simply put, there is no state. By analogy to how a tree space treats a child tree space as a mere value, independent from its concurrent evaluation, it also doesn't give rise to state beyond the child tree space, as again it is treated as a non mutable, functional value. Concurrency and state are made possible through the same features, and it is by the same mechanism (shielded tree spaces) that state doesn't have any effect on higher levels.

Besides shielding of tree spaces towards their environment, a crucial feature in all of this is the linearity of all data structures involved. The following holds in Aardappel: *any runtime value (tree or tree space) has exactly one parent at any point during its lifetime* (see also the next chapter). If there is exactly one parent of a tree space, and that parent regards the evaluation to normal form of the tree space as one reduction step (as is the case in Aardappel semantics), there is no notion of state outside of the tree space where the state is present. Similarly for a tree inside a tree space, linearity guarantees that it is either directly part of the tree space (in which case no one has access to it), or it is consumed by one other tree (in which case only that tree has access to it, until and if it feels like making it available again by using `out`).

So to understand Aardappel's sharing model, we have to look at tree space semantics from two perspectives, from the inside and the outside. From the inside (and only directly inside it, not from a nested tree space) trees can make use of concurrency and state, but are perfectly shielded and safe to access. From the outside a tree space is a purely functional bag data structure. Put this together and you have a sharing model that has *state without shared mutable data structures*: the only thing that is ever mutated is the tree space, and as we saw a tree space is never shared, it is linear[7].

4.3.3 The set of observers

We introduced the metric of the "set of observers" in chapter 2.3 to be able to compare sharing models more precisely, and to show where possible dependencies are coming from. Besides the number of observers, their locations are also important in determining how dependencies can undermine the programmer's ability to cognitively keep an overview over a program: if multiple observers are all gathered in the same local

[7]One might object that a tree space is shared by the trees within it, but this is not the case. The trees in a tree space don't have access to the tree space at all, only to their neighbouring trees. And neighbour trees are only shared in a weak way as at most one process can access the value at one time, and these value are never mutated (merely consumed and replaced).

construct, it should be much easier to track down dependencies than when they are scattered randomly throughout a large program.

This is exactly the property Aardappel has: *the set of observers of a piece of state corresponds 1:1 with the tree space*. The entities that have access to a particular tree in a tree space are exactly all the other trees in that tree space, and nothing else (not even nested tree spaces), which greatly reduces the complexity of dependencies. This is why Aardappel's sharing model can be viewed as somewhere along the line between "any imaginable set of observers" (imperative) and "an empty / singleton set of observers" (functional): it has the set of observers as an explicit language feature, i.e. the tree space. The idea here is that something like access to state is of such importance, making it explicitly and precisely viewable at the language level helps the programmer to keep total control (compared to imperative programming) while opening up more ways of structuring programs (compared to functional programming).

The design philosophy of this model is similar to functional languages: starting with a restrictive model will promote bug-free programming in its most natural use (i.e. without having to adopt a certain style), as a model that only *potentially* allows bug-free programming (if it's used in "the right way") is of no use[8]. The difference is that Aardappel extends the singleton set to the tree space, with great consequences.

4.4 Graphical syntax

Aardappel's default syntax is a graphical one, and indeed the only syntax that is available in its implementation. The reasons for making Aardappel a graphical language were simple: there was a fairly natural mapping for it to a graphical representation, and also the added features make it interesting research in its own right.

But for most languages you have to ask yourself wether a graphical syntax is a good idea at all. People generally have the intuition that "graphics must be superior to text", and in the case of graphical languages that is a statement that is a real struggle to try and make come true [50]. There are many properties of textual syntaxes that make it a formidable opponent, and especially for general purpose languages there are some hurdles that are almost impossible to overcome for graphical languages. Special purpose languages fare a bit better because their often limited syntax and / or semantics can

[8]This is also why OO along this metric is really no different than regular imperative languages. It may promote reduction of dependencies by clever use of data hiding, but on the semantic level there are no restrictions to the structure of the set of observers. Programming is still based on graphs of shared mutable data structures, unlike Aardappel and functional languages.

be exploited to design a good graphical representation, and they are not expected to be used in complex situations. Aardappel is a general purpose language, but both its syntax and semantics are minimalistic and uniform.

4.4.1 The first hurdle: structure

Textual languages structure code along a single dimension using nesting, which can be viewed as the one dimensional version of a tree structure. All that can't be expressed using nesting (i.e. needs a graph) is patched up using identifiers. Although seemingly clumsy, this system has great advantages because identifiers can express any complex graph without cluttering up the screen. Both nesting and identifiers create structure without making it graphically obvious (even though nesting can be used effectively with indentation to help spot the relations). To call a syntax graphical it should ideally use real graphical relationships to replace these. Most programming languages however look like a graph when using their most natural graphical representation, and so far it has been very hard to avoid graphs becoming a mess on screen.

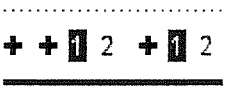
Aardappel uses a tree structured representation, not a graph. To some extend this is not a real solution, as we are only side-stepping the problem: every general purpose language has features best represented as at least a directed acyclic graph (DAG), and choosing a tree representation means the burden is shifted elsewhere. This design decision was taken because a graph representation was not deemed acceptable, and we saw new ways of solving the remaining problems. Besides this, Aardappel is a language that is all about trees, so this graphical mapping is very natural, unlike say representing a dataflow language using trees. Trees give us many advantages that make actually editing this syntax very comfortable: trees can be automatically laid out in 2d space to great perfection, while utilising screen real estate better than graph based syntaxes.

There are multiple ways of removing situations where a DAG would be needed because of a shared sub-expression, and the most basic way to do that in the graphical syntax is by using local rules. This corresponds to the following transformation:

$$...sharedexp...sharedexp... \Longleftrightarrow \lambda(sharedexp)\ \texttt{where}\ \lambda(x) = ...x...x...\ \texttt{end}$$

This is similar to the `when` syntactic sugar we introduced earlier. This may seem awkward but in the visual version it is often most convenient, because it allows for additional pattern matching, which is often a reason on its own to use a local rule. An even

more visual way of sharing is part of Aardappel's graphical syntax, that allows simple sharing of sub-expressions, which are kept in tree form by duplicating it and giving it a different colour background. All operations on that sub-expression are shared as well, the duplication is only the representation.



It is debatable whether this is really an improvement over just using local rules directly (like the example code shown in chapter 3 does).

4.4.2 The second hurdle: placeholders

Another feature of identifiers which people take for granted in textual programming but is a functionality that is almost impossible to replace is that identifiers are self documenting. If I read the code for a `qsort` function that features identifiers such as `pivot`, `tail`, and `partition`, their repeated occurrences guide me as to the meaning of the code. But if all of these are anonymous black lines, I will have to stare much harder to follow what values are used for what purpose, given that I understand what the code is supposed to do in the first place. If the code is so big I have to look back and forth between code fragments, that black line is not going to help me remember what value was flowing along it, but `pivot` will. A further advantage of identifiers is that they are easy to pattern match for the eye, it is somehow easier to spot that two words are the same than say two colours, or even two icons.

In the previous section we saw the way Aardappel is tree structured and its use of local rules to express sharing shifts all need for identifiers towards placeholders, i.e. variables in a pattern. Aardappel's use of example values instead of identifiers is arguably the only bit of syntax that can compete with identifiers when it comes to allowing for easy visual recognition of the values, and their self documenting value (People generally prefer to think in terms of examples rather than abstractions, see e.g. [44]). Example values often go much further than identifiers in documenting how the code works, compare the `append` example with its textual equivalent, which looks very abstract in comparison: in the visual version you can actually spot immediately how the elements of the two lists are grouped and shuffled around. Actually creating the examples is usually easier than coming up with a new identifier if the value has some structure to it, however for simple values like integers and booleans example values often are less informative than identifiers.

4.4.3 *Editing*

A great advantage of a graphical syntax, is that in the editing process the environment can be much more helpful and intelligent, and provide an a large amount of features for editing, debugging and structuring the code. The editing process itself can be used as a hint to the environment how to treat the code, in the case of Aardappel the dragging of trees from the pattern to the expression which automatically makes values into place holders. This speeds up editing a lot by saving the user from having to assign these properties manually.

Aardappel opens up all sorts of new ways of debugging as well, in multiple steps: first of all the code is already more insightful thanks to the use of examples as placeholders. Second, quick folding and unfolding can give further feedback on what the code does by viewing intermediate results of sub-expressions, something which can easily be done while constructing code. Normal evaluation of any program fragment can happen at any point during construction, all because of the property that code is always complete at any point. Because everything is a tree, even the output of your programs, it is easy to reuse bits of output or intermediate trees halfway through an evaluation as part of the program[9]. All this blurs the distinction between editing and debugging which has become almost a continuum, unlike classical environments where editing is separated by edit - compile - run - supply example input, and repeat.

Besides making meaningful placeholders and supplying a default value for evaluation, examples also supply a default value for creating new trees. Whether I create a new call to `append` by dragging an existing tree or just typing `append`, I will get a full tree, and the child values will give me an idea of how the function is to be used. In some cases it will even save me modifying a child as the example provides a sensible default value. For `append` this isn't that useful, but for more complex trees this can really speed up editing. Because you can always supply example values to new trees you define, there is a lot less strain on the trade-offs between simplicity and functionality: trees can be made more complex without a penalty in usability.

[9]With help from the environment, partial evaluation type functionality could fit into this quite seamlessly.

Chapter 5

Semantics Specification

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> – *Donald Knuth*

5.1 The specification

The semantic specification has been written in an operational semantics style. The reduction semantics of the core language elements takes the form of a set of non deterministic rewrite rules, which make use of a set of functions (most of which are predicates, i.e. functions with a boolean result type). For the most part the specification tries to adhere to existing standards (of logic etc.), though some of the syntax used may need additional clarification.

5.1.1 Types

The types used are similar to what you'd find in a functional language (e.g. SML): $Type \rightarrow Type$ is a function type, $Type \times Type$ is a Cartesian product (tuples, triples etc.), $[Type]$ and $\{Type\}$ are list of or multiset of $Type$ respectively.

5.1.2 Multisets

The specification uses multisets throughout instead of sets. Using the familiar $\{\}$ syntax, a multiset is defined as a total function which maps any value on to its number of

occurrences in the multiset ($s : Any \rightarrow Nat$)[1]. We can therefore define the multiset operations used in this specification as follows:

$$
\begin{array}{lll}
x \in s \quad \text{(multiset membership)} & = & s(x) > 0 \\
s \ominus x \quad \text{(multiset subtraction)} & = & \begin{array}{ll} s \text{ except } x \mapsto s(x) - 1, & \text{if} s(x) \geq 1, \\ s, & \text{otherwise} \end{array} \\
s \oplus t \quad \text{(multiset concatenation)} & = & \lambda x.s(x) + t(x) \\
f(\!|s|\!) \quad \text{(multiset map)} & = & \lambda y.(\Sigma x | f(x) = y.s(x))
\end{array}
$$

5.1.3 Lists

Lists (ordered multisets if you will) are denoted by the familiar [ ] syntax, where [ ] is an empty list and $[h|t]$ is a constructor for a list with a head and a tail list (of polymorphic type $A \times [A]$). Similar to multisets there is the usual list map $f(\!|l|\!)$ which applies f to all elements of l.

5.1.4 Pattern matching

The underscore character (_) is the anonymous variable, i.e. it has the same semantics as a unique variable which is not referenced again, and has no name introduced. x/y (an *as-pattern*) denotes that both x and y will be matched onto the same expression: the match will only succeed if both match, and bindings introduced by both will be valid. For example:

$$[h|\_]/x \longrightarrow [h|x]$$

This rewrites as list like `[1,2,3]` into `[1,1,2,3]`.

5.1.5 The rules

All the rewrite rules in section 5.4 rewrite an expression and an environment together, denoted by the syntax $Exp : Env$. A rewrite rule specifies one possible reduction step in the evaluation expression (denoted by $\longrightarrow$). Its general format is:

$$\frac{\begin{array}{l} booleanexpression1 \\ booleanexpression2 \\ \ldots \end{array}}{exp : env \longrightarrow exp' : env'}$$

[1] if the range was restricted to just 0 or 1 we would call it a set

Note that we allow bindings introduced by exp and env to be used by the boolean expressions, and bindings the boolean expressions introduce to be used in exp' and env'. A rule like this should be read as: we can perform a one step reduction of $exp : env$ to $exp' : env'$ if there is any set of bindings such that the boolean expressions can be satisfied and suitable bindings exist.

The meaning of any expression (or program) in the language is a normal form with respect to these rules, more formally, the meaning of an expression x is an expression x' where the following holds[2]:

$$\exists i \geq 0.x \longrightarrow^{i} x' \wedge \neg(\exists x''.x' \longrightarrow x'')$$

where

$$\begin{aligned} x \longrightarrow^{0} x &= true \\ x \longrightarrow^{i+1} x' &= \exists x''.x \longrightarrow^{i} x'' \wedge x'' \longrightarrow x' \end{aligned}$$

Note that for one particular x, there is a non empty set of x' for which this holds. This expresses the correct semantics of Aardappel, as it is a concurrent language whose real evaluation is non-deterministic as well, i.e. repeated runs of *some* programs may give different results. A particular evaluation of x can be any concatenation of valid one step reductions to reduce x to x', not necessarily an optimal one[3].

A trivial example that can reduce in different ways is the bag `{ a(1) a(2) b(1) }`, which, using the rule:

```
a(X) { b(Y) } = X+Y
```

can result in either `{ a(2) 2 }` or `{ a(1) 3 }`.

A program (i.e. a top level expression) almost always needs to be a tree space to be meaningful, because a tree expression that performs Linda operations without a parent

[2]Technically, this is only valid for terminating programs. As mentioned in chapter 3, live lock in Aardappel does not preclude that the program is doing a useful job. The specification in this chapter only specifies the meaning of terminating programs.

[3]This is where Aardappel differs greatly from languages like Gamma, whose semantics consist of similarly non deterministic reduction rules, but which guarantees the optimal route to x', of which there is exactly one. The implementation of this requires backtracking.

tree space is meaningless. For a program that uses Linda operations, there is not necessarily a single tree as the result of a tree reduction: the result of reducing a tree x that has been wrapped to $\{x\}$ for evaluation may be a tree space with more than one tree[4]. As will be obvious below, the semantics of trees is only specified within the context of an environment.

5.2 The abstract syntax

This grammar specifies the structure and types of the core language constructs used in the semantics below. It has virtually the same structure as the grammar of chapter 3, except that we merge expressions and patterns into $Tree$. Env and $Binds$ are needed to model the runtime information that needs to be present during an evaluation.

$$
\begin{array}{lcl}
Rule & = & rule(Tree, [Exp], Exp) \\
Tree & = & node(Atom, [Exp]) \\
Bag & = & bag(\{Exp\}) \\
Atom & = & atom(Ident) \\
Exp & = & Tree \mid Bag \mid var(Ident) \mid out(Exp, [Exp]) \mid int(Int) \\
Env & = & Binds \times [Rule] \times [\{Exp\}] \\
Binds & = & Ident \rightarrow Exp
\end{array}
$$

Rules consist of a left hand side and a right hand side, the list of expressions in between are the Linda `in` patterns (they are specified as being part of a $Rule$, since being part of an Exp such as the Linda `out` would be overly generic[5]). $Binds$ are modeled as a mapping from identifiers to expressions. $[Exp]$ in out are the expressions to be put in the surrounding tree space, where Exp is the resulting expression. An environment Env consists of a list of bindings, the set of rules, and a list (stack) of tree spaces (to correctly handle hierarchical tree spaces).

A static program consists of a Bag and a $[Rule]$, a Bag is evaluated in the context of an Env which other than $[Rule]$ is initially empty. The $[Rule]$ is statically preprocessed

[4]Operationally it is technically possible to keep track of the original tree in the tree space and denote that as the result of a evaluating an unwrapped tree. This however would be extremely unnatural as the normal form of the original tree may not even be present anymore once the tree space reaches normal form. Besides that, it would introduce a very operational notion of tree identity into the semantics which would unnecessarily complicate things.

[5]This non uniform way of doing things makes sense given their respective properties: if a Linda `in` operation wasn't tied to a rule the semantics of the language would be much more complicated with no gain in expressivity. The `out` operation however has very simple semantics and can profit from the flexibility of being able to appear at any level in any expression (for explicit sequencing).

to be sorted on specificity, i.e. from most specific to least specific. This simplifies the following specification a great deal as it can simply assume that the first rule out of this list to match is the correct one (specificity ordering rules for Aardappel were informally described in chapter 3).

5.2.1 Local rules

Local rules, though part of Aardappel, have not been specified in this semantics, the reason being that they are transformed to global rules statically[6]: they are not part of the core language. In this section we give an example how this works. One can imagine a local rule adding the following to the abstract syntax above:

$$Exp = local(Exp, [Rule]) \mid \ldots$$

A *local* is transformed to just an Exp and an extra global $Rule$. example:

```
f(x) = g(3) where g(y) = x+y
```

g is a local rule. This is transformed to:

```
f(x) = f_g(3,x)
f_g(y,x) = x+y
```

This includes two trivial transformations: local rules are made into global rules using new unique identifiers, and any free variables are passed as extra arguments, which fully preserves the intended semantics. It is the transformed version which the above specification specifies the semantics for.

5.3 Functions

These functions specify essential properties about expressions, which are used in the rules in section 5.4. All but one of these functions take the form of a predicate, i.e. a function returning a boolean.

[6]This unfortunately does not cover local normal forms (section 3.1.2), which aren't specified here at all.

$$\text{NORMAL} \quad : \; Exp \rightarrow Env \rightarrow Bool$$

$$\begin{array}{lll}
\text{NORMAL} & int(_)\ _ & = \; true \\
& var(_)\ _ & = \; false \\
& atom(_)\ _ & = \; true \\
& out(_,_)\ _ & = \; false \\
& bag(l)\ e & = \; \forall x.x \in l \Rightarrow \neg \text{ACTIVE}\ x\ e \\
& node(h,c)/n\ \ e & = \; \text{NORMAL}\ h\ e \wedge (\forall x.x \in c \Rightarrow \text{NORMAL}\ x\ e) \\
& & \quad \wedge \neg(\exists e', r.\text{FIND}\ n\ e\ r\ e')
\end{array}$$

NORMAL tells us for an expression in a certain environment whether it is in normal form or not. A Bag is in normal form if all its elements are in normal form, or, as a special case, if they are all blocked. A $Tree$ is in normal form if its head and all children are in normal form, and no rule applies to it (expressed by FIND , see its definition below).

$$\text{BLOCKED} \quad : \; Exp \rightarrow Env \rightarrow Bool$$

$$\begin{array}{lll}
\text{BLOCKED} & node(_,c)/n\ e & = \; (\exists l, e', e'' : \text{FIND}\ n\ e\ rule(_,l,_)\ e' \wedge \neg(\text{IN}\ l\ e'\ e'')) \\
& & \quad \vee \exists x.x \in c \wedge \text{BLOCKED}\ x \\
& \text{otherwise} & \; false
\end{array}$$

BLOCKED expresses the property that the expression can only be evaluated if it can successfully retrieve tree(s) from the tree space, and until it can do so it is said to be BLOCKED . As Linda `in` is associated with a rule, only a $Tree$ can be blocked. A tree is blocked if there is a rule that can be applied to it (again, FIND expresses this), and that rule requires tree(s) to be taken from the tree space which aren't there (expressed by IN). It is also blocked if any of its children are blocked.

$$\text{ACTIVE} \quad : \; Exp \rightarrow Env \rightarrow Bool$$

$$\text{ACTIVE} \quad x\ e \; = \; \neg \text{NORMAL}\ x\ e \wedge \neg \text{BLOCKED}\ x\ e$$

For a $Tree$, NORMAL and BLOCKED are mutually exclusive properties[7], and so ACTIVE defines the third state a $Tree$ can be in. For a $Tree$ to be active (in its context) means it's not done reducing and it's also not blocked. Since for everything besides a $Tree$ BLOCKED is never the case, ACTIVE is always equivalent to $\neg$NORMAL.

$$\begin{array}{lll}
\text{FIND} & : Tree \rightarrow Env \rightarrow Rule \rightarrow Env \rightarrow Bool & \\
\text{FIND} & x\ (_, r, _)/e\ y\ z & = \text{FINDR}\ x\ r\ e\ y\ z \\
\text{FINDR} & _\ []\ _\ _\ _ & = false \\
 & x\ [rule(y, _, _)/r|l]\ e\ r'\ e' & = \text{MATCH}\ x\ y\ e\ e' \wedge r = r' \\
 & & \vee\ (\neg(\exists e''.\text{MATCH}\ x\ y\ e\ e'') \\
 & & \wedge \text{FINDR}\ x\ l\ e\ r'\ e')
\end{array}$$

FIND is split in two functions for simplification reasons. FIND is true if it can find a rule for the $Tree$ such that it is equal to the $Rule$, and the bindings that match its left hand side with the $Tree$ will create are those that make the first Env into the second. The style of writing this function is such to accommodate their non deterministic usage in the rules, and can more easily be read if one considers the last two arguments, i.e. $Rule$ and Env as "return values". Many of the functions that follow have been written in a similar style. Notice how the function enforces that only the first rule that matches can be a valid result: either the current rule matches, or it doesn't and any of the following may (the order of the list of rules is important because it is pre-sorted on most specific rule first).

[7] Trivially, because they require the opposite outcome for FIND each.

$$\text{MATCH} \quad : \; Exp \rightarrow Exp \rightarrow Env \rightarrow Env \rightarrow Bool$$

$$\begin{array}{lll}
\text{MATCH} & int(n)\ int(n')\ e\ e & = (n = n') \\
& atom(i)\ atom(i')\ e\ e & = (i = i') \\
& node(a, [])\ node(a', [])\ e\ e & = \text{MATCH}\ a\ a'\ e\ e \\
& node(a, [h|t])\ node(a', [h'|t'])\ e\ e'' & = \exists e'.(\text{MATCH}\ h\ h'\ e\ e' \wedge \\
& & \quad \text{MATCH}\ node(a, t)\ node(a', t')\ e'\ e'') \\
& bag(m)\ bag(m')(b, r, s)(b', r, s) & = \exists l''.\text{bagify}(l'') = m' \\
& & \wedge \text{IN}\ l''\ (b, r, [m|s])\ (b', _, _) \\
& x\ var(i)\ (b, r, s)\ (b', r, s) & = \text{if}\ (b\ i) \neq \bot \\
& & \text{then}\ b' = b \wedge (b\ i) = x \\
& & \text{else}\ b' = \\
& & (\lambda a.\text{if}\ a = i\ \text{then}\ x\ \text{else}\ b\ a) \\
& \text{otherwise} & false
\end{array}$$

MATCH tells us whether an expression and a pattern can be unified in the context of a certain Env and result in another Env (arguments in that order). An interesting case is the one for var which introduces a new binding, but only if the variable wasn't already bound, in which case the two values must match (structural equivalence)[8]. IN, used in the case for bag, is defined below, and succeeds if it can take a set of trees out of another set.

$$\text{CONSTRUCT} \quad : \; Exp \rightarrow Env \rightarrow Exp$$

$$\begin{array}{lll}
\text{CONSTRUCT} & int(n)\ _ & = int(n) \\
& var(i)\ (b, _, _) & = b\ i \\
& atom(i)\ _ & = atom(i) \\
& node(a, l)\ e & = node(\text{CONSTRUCT}\ a\ e, (\lambda x.\text{CONSTRUCT}\ x\ e)\ (\!|l|\!)) \\
& bag(l)\ e & = bag((\lambda x.\text{CONSTRUCT}\ x\ e)\ (\!|l|\!)) \\
& out(x, l)\ e & = out(\text{CONSTRUCT}\ x\ e, (\lambda x.\text{CONSTRUCT}\ x\ e)\ (\!|l|\!))
\end{array}$$

CONSTRUCT is the only non predicate definition, and is a straight mapping from Exp to Exp, replacing variables by their bindings as it goes.

[8]The case for node suggests ordering, yet it isn't easy to write differently as some order needs to be present to pass new environments around. It also would need zip and unzip.

$$\text{IN} \quad : \quad [Exp] \rightarrow Env \rightarrow Env \rightarrow Bool$$

$$\begin{array}{lll}
\text{IN} \quad [] \; x \; x & = & true \\
\quad [h|t] \; (b, r, [l|s]) \; (b', r, [l'|s]) & = & \exists x, b'.x \in l \wedge \text{NORMAL } x \\
 & & \wedge \text{MATCH } x \; h \; (b, r, [l|s]) \; (b', r, [l|s]) \\
 & & \wedge \text{ IN } t \; (b', r, [l \ominus x|s]) \; (b'', r, [l'|s]) \\
\quad \text{otherwise} & & false
\end{array}$$

IN is true if it can match a list of patterns from the topmost tree space in the first Env and the resulting bindings would be equal to those added to the second Env. For this to work there obviously have to be more trees in the top level tree space than in the list of patterns. Matching the patterns is ordered and non-backtrackable, i.e. they can be thought of as sequenced single tree IN operations at a time, even though the whole is an atomic operation. Actually selecting a tree from the tree space is unordered, any out of the set of matching trees (should there be more than one) can be selected. The result of this is that it is possible to perform an IN operation which fails (and causes the parent tree to block) even though a good match is possible, for example given this rule:

```
a(x) { b(y) b(c(z)) } = x+y+z
```

and a tree space like:

```
{ b(1) b(c(1)) a(1) }
```

the `a(1)` tree will try to match the `b(y)` pattern first, but there's no guarantee it won't match `b(c(1))` (as tree spaces are unordered), and thus cause the whole IN operation to fail.

5.4 Rules

These rules specify the reduction steps for trees and bags as mentioned above (most other expressions are in normal form straight away). The order of evaluation is specified in a completely non deterministic way. Not many constructs in the language need an order of evaluation, and where it's needed (because of state) it is imposed by the tuple space, i.e. by the fact that trees are one of NORMAL , BLOCKED or ACTIVE .

Evaluation starts with $b : ((\lambda\_.\bot), r, [])$, where b is the tree space to be evaluated and r the list of rules.

$$\frac{\begin{array}{l} x \in l \\ \text{ACTIVE } x \\ x : (b, r, [l \ominus x | s]) \longrightarrow x' : (\_, \_, [l' | \_]) \end{array}}{bag(l) : (b, r, s) \longrightarrow bag(l' \oplus \{x'\}) : (b, r, s)}$$

This rule governs the evaluation of bags, it basically says: if you can find a tree in the bag that has the ACTIVE property, perform any number of reduction steps it. If it can't find any trees to reduce, effectively no rules apply to this bag, and it will be in normal form. This includes the situation when all trees are BLOCKED . An implementation could use any method to pick a tree for reduction out of a tree space (i.e. what is specified by $x \in l \wedge \text{ACTIVE } x$), with the restriction that it has to be *fair*[9]: every active tree in the tree space gets picked in a finite amount of reductions steps, i.e. given two active trees it can't just keep reducing one tree infinitely and starve the other (no such traces are allowed). This also ensures that some potentially non terminating programs (in a starvation scenario) are guaranteed to terminate (and thus be covered by this specification), for example:

```
a(x) { b(y) } = { b(y) } a(x+y)
c(x) { b(y) } = c(x+y)
```

Given a bag `{ a(1) b(1) c(1) }`, `a(x)`could evaluate infinitely, but fairness ensures this program terminates.

$$out(x, l) : (b, r, [l' | s]) \longrightarrow x : (b, r, [l \oplus l' | s])$$

The reduction of an *out* is simply to add its expressions to the topmost bag in the environment.

$$\frac{\begin{array}{l} i \in 0..\#l - 1 \\ \text{ACTIVE } l[i] \\ l[i] : e \longrightarrow c : e' \end{array}}{node(a, l) : e \longrightarrow node(a, l[i \leftarrow c]) : e'}$$

[9]For simplicity's sake not specified formally here. Defining fairness more formally could be done as a property of infinite traces of reduction steps (see 5.1.5).

This rule allows an (active) child of a tree to reduce itself. Note the special replacement syntax we use: $l[i \leftarrow c]$ means l but with the element at i replaced by c. This is necessary to ensure that a reduction step of a child replaces just the child that was reduced, to preserve the semantics of tree space operations. This rule also implicitly encodes the linearity properties of tree spaces: if $l[i]$ is a tree space child, the parent tree will never witness its concurrent and stateful reduction as from the perspective of this rule it is being reduced in one step to a (normal form) tree space value.

$$\frac{\begin{array}{l}\forall c : c \in l \Rightarrow \text{NORMAL } c \\ \text{FIND } \mathit{node}(a, l)\ e\ (\_, t, x)\ e' \\ \text{IN } t\ e'\ e''\end{array}}{\mathit{node}(a, l) : e \longrightarrow \text{CONSTRUCT } x\ e'' : e''}$$

This rule evaluates the tree itself, but only if all its children are done evaluating (to enforce eager evaluation order). The tree can only reduce if there is a rule for it (guaranteed by FIND), and we can get the needed trees from the tree space (guaranteed by IN). The conditions are actually the reverse situation from the definition of BLOCKED . Upon reduction, CONSTRUCT replaces all bindings by their values, making use of bindings introduced by FIND, and IN.

Chapter 6

The implementation

This chapter discusses the Aardappel implementation.

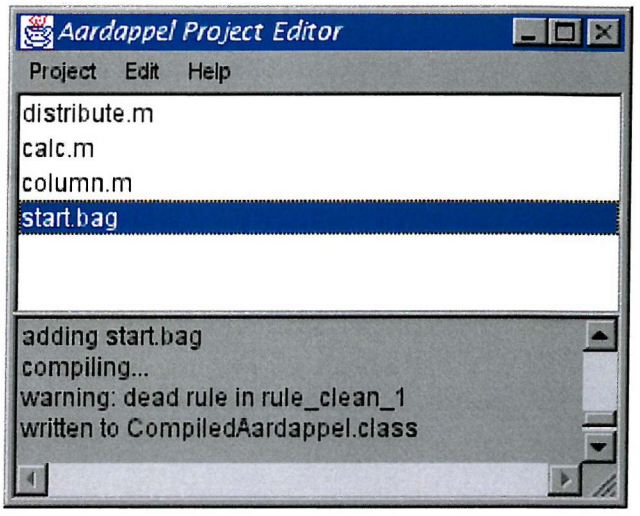
6.1 Implementation choices

Even though the core of this research, Aardappel's sharing model, is of a theoretical nature, part of the plan has always been to provide a fully featured implementation. Many theoretical advances in language design have little value on their own: their value is that they alter the way in which the programmer structures a program, and as such have influences on the cognitive process of programming. To be able to evaluate a language design we must therefore be able to experience it as a practical programming language, and to test it accurately we need to make a full implementation that does not have any of the limitations a prototype implementation usually has. In Aardappel's case this requirement was taken even further, as we wanted to try the visual syntax in action, and exploit Aardappel's semantics by using a fully distributed runtime system.

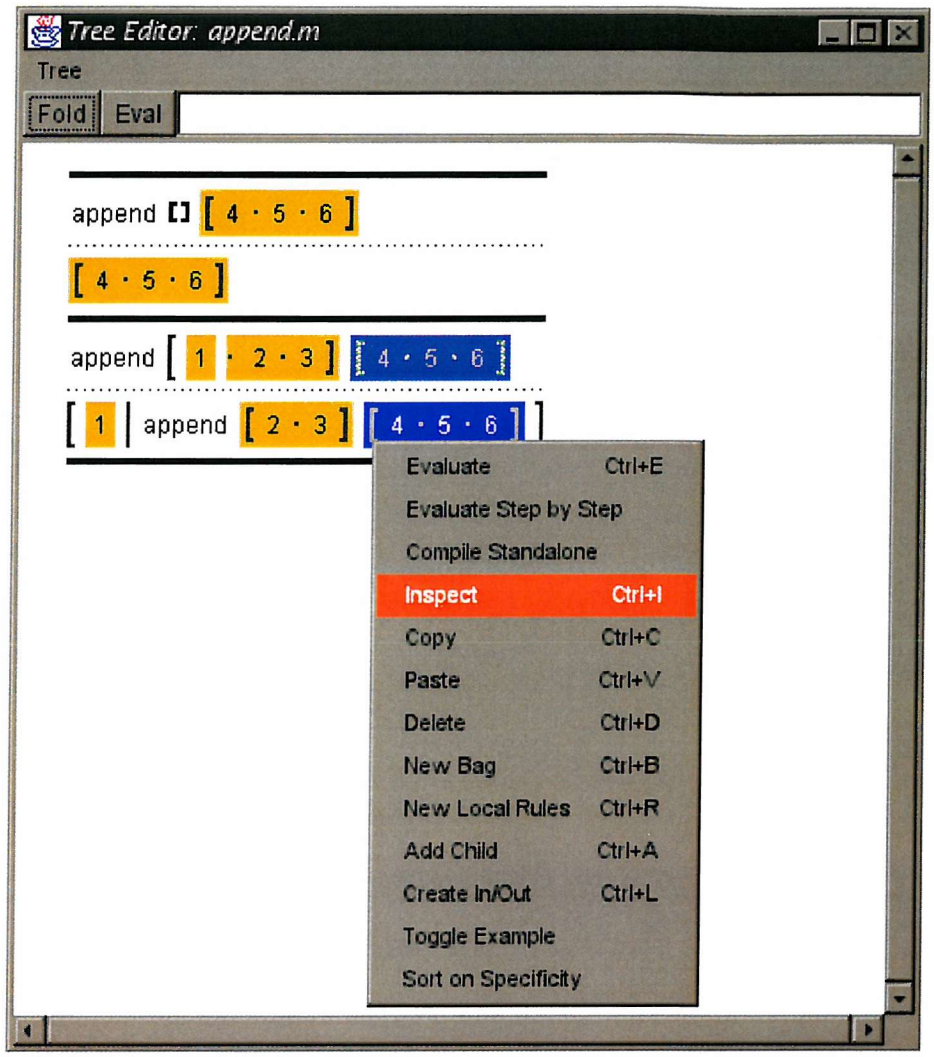
Java was chosen as the platform for both the programming environment and for compiled Aardappel programs. The reason was particularly simple: because all required technologies (graphical interface, code generation and loading, and Remote Method Invocation (RMI)) were available within one system. The following sections will discuss all components in turn, and mention what influence the choice of Java had on their implementation.

6.2 The IDE and graphical editor

The main window of the implementation is the project editor. An Aardappel project consists of a set of modules (each of them a set of rules) and a set of bags, which are various main programs (tree spaces), making use of the rules. The project manager allows you to add/remove items to the project as you expect, and open multiple graphical editor windows.



Double-clicking on any of the items in the list opens a tree editor for that particular item. Tree editor windows are used to edit modules, bags, and in general any piece of Aardappel code: you can select any sub expression and edit it in a separate window (inspect). Evaluation results are show in tree editor windows as well and can be used for editing.



There is one main way of editing code, and that is drag and drop. Drag and drop copies a (sub)tree from any rule in the window to any other place. By choosing suitable examples, it is usually possible to create new values solely by drag and drop, starting with the root of a new tree and replacing children as you go.

An important side-effect of drag and drop is that the editor uses it to infer what the relation is between the two copies after the drag and drop, i.e. it matters where the source of the destination lie. By default, if there is no particular relation (i.e. if source and destination span two rules), the result is a copy, and the two trees will be unrelated from there on. One of the most useful situations is where the source of a drag and drop is in the pattern part of some rule, and the destination is in the corresponding expression part: this creates a placeholder. Placeholders, as shown before, are marked with an orange background, and the two trees are shared, i.e. if you modify something within the tree it will show up in both copies. To check which trees are shared (besides the fact that they look visually identical) you can single-click on a tree which will highlight

it and all its shared copies in a darker colour (this is shown in the screenshot above: the lower `[4,5,6]` in the second rule is the expression selected, and the one above it is marked as being identical with it). Dragging within one expression can create a general shared copy (similar to a let-expression).

An alternative way of creating new trees is typing the name of the atom at the top of the window: the editor will then look up the most specific tree it can find matching this atom in any of the modules, and put this in place. This way, you will have most of the structure in place to work from, and an example of what the tree should look like. For syntactical elements that cannot be created by typing or dragging, there are several menu options to create them (bags, local rules, in/out operations).

One of the most useful operations in the editor is instant evaluation / testing of any piece of code: simply right-click on any expression and select "evaluate" to see what it evaluates to. As mentioned before, since no expression is ever incomplete, this can almost always be done.

6.3 The compiler

Part of the graphical development environment is the compiler, which is used to implement both the "evaluate" and the "compile stand-alone" operations. The compiler compiles the internal structure of all modules part of a project and the expression selected for evaluation to bytecode for the Java virtual machine, in the form of a .class file. In the case of "evaluate", the compiler compiles to a bytecode array in memory which is then passed to a custom classloader which verifies the bytecode and executes it on its own thread. This means that using "evaluate" has an interactive and almost instant feel to it since no file system access is needed and the current compiler is very fast. "compile stand-alone" does the same thing but instead of feeding the output to the classloader it writes a .class file, which can then be executed independently from the graphical editor and compiler (just a copy of the runtime system classes is required).

The compiler does a certain amount of optimizations to make tree rewriting reasonably fast and cut down on garbage collection overhead: it groups rules for the same atom and arity into (static) methods to factor out pattern matching. These methods double as constructors if none of the rules apply. Expressions are then compiled into calls to these methods which means that no intermediate tree structures for outermost trees are ever created. Normal form trees are mapped onto Java objects. Bag expressions and out operations are compiled to separate methods which take a list of free variables. They are

associated with a unique id and called from a global table on demand (this is necessary because each of them is launched as its own thread and as such out of context of its parent). Free variables in general are translated to extra arguments, where possible. Pattern matching for `in` operations is optimized to O(1) lookup for the top level atom (using indexing into arrays), this means that in the average case where an `in` operation is a simple tree with variables as children, Linda operations should be blindingly fast. The actual pattern matching is a complex "conversation" with the runtime system that makes sure that a single `in` operation can atomically grab multiple trees without other threads interfering.

As a simple example of the code that the compiler generates, let's look at the compiled version of `append`[1]:

```
append([],y) = y
append([h|t],y) = [h|append(t,y)]
```

Both rules are compiled to a single method, the second rule starts at label `L18`. The first rule takes the first argument and looks at its `id` field (this field contains a unique integer corresponding to an atom/arity pair, `Rval` is the superclass of all tree objects), if it is not equal to the id for `nil` then it branches to the second rule. If it is, it returns the second argument (as is obvious from this code, the compiler hardly does any low level optimisations).

```
method public static final rule_append_2(Lr/Rval;Lr/Rval;)Lr/Rval;
  aload_0                                  // get first arg
  dup
  getfield          r/Rval/id I
  bipush            12                     // id of ''nil''
  if_icmpne         L18                    // is head atom == nil ?
  checkcast         r/Rval
  pop
  aload_1
  astore_2
  aload_2                                  // return y
  areturn
L18:
```

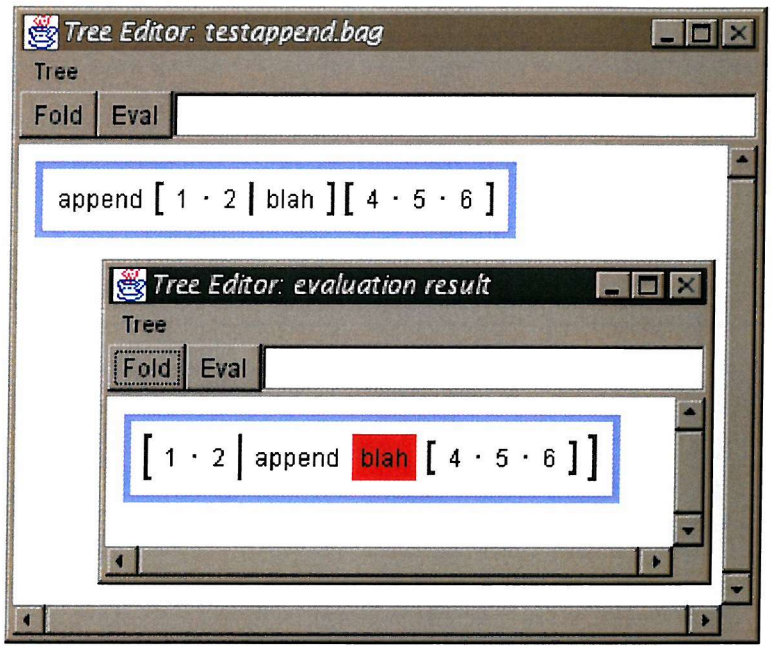
[1]This doesn't use any of the more interesting features such as threads, in/out operations etc. but to demonstrate these properly long bytecode listings would be necessary which seems inappropriate for the current purpose.

```
   pop
   aload_0
   dup
   getfield        r/Rval/id I
   bipush          27
   if_icmpne       L57                  // is head atom == cons ?
   checkcast       r/Rtree2             // cast for further access
   dup
   getfield        r/Rtree1/c1 Lr/Rval; // grab h
   astore_2
   dup
   getfield        r/Rtree2/c2 Lr/Rval; // grab t
   astore_3
   pop
   aload_1
   astore          4
   aload_2
   aload_3
   aload           4                     // call append and cons
   invokestatic    CompiledAardappel.rule_append_2
                                   (Lr/Rval;Lr/Rval;)Lr/Rval;
   invokestatic    CompiledAardappel.rule_cons_2
                                   (Lr/Rval;Lr/Rval;)Lr/Rval;
   areturn
L57:                                 // if append is called
   pop                               // with trees other than
   new             r/Rtree2          // nil or cons, create
   dup                               // new append normal form
   bipush          26
   aload_0
   aload_1                           // call constructor
   invokenonvirtual r/Rtree2.<init>(ILr/Rval;Lr/Rval;)V
   areturn
end method
```

The second rule does something similar, and checks if this is a cons tree. If that isn't the case, it branches to L57, which is the code that turns append into a normal form

tree by creating a new object. If this is a `cons` tree, we cast the value to a subclass of `Rval` with 2 children, `Rtree2`, and read them both into local variables (the casting is necessary to please the JVM class verifier). We then call respectively `append` and `cons` on the values obtained and return the result. If you create an `append` tree with children that are not either `nil` or `cons`, the code below creates a new `Rtree2` object as normal form.

One feature where the compiler cooperates with the editor is in providing visual feedback of analysis. Much like in ProLog, in tree rewriting it is often difficult to make out whether something is an error or actually meant that way, for example if there is no rule to rewrite a certain tree it is not necessarily an error. The compiler, as part of its general analysis for optimisation, analyses the contexts that trees are used in and provides hints to the editor. When rendering the result of an evaluation the editor then marks these, for example it marks in red the first tree that wasn't reduced but the compiler estimates was intended to be evaluated (because it reduced to a normal form yet there exists no rule that uses it as a normal form).



This is what happens if you try to append a list which ends in `blah` rather than `nil`: the compiler points out the likely culprit. So far this mechanism has almost always been right, and turns out to be a very valuable tool.

Efficiency of sequential code overall is bearable, for code that is heavy on processing complex data structures speed is roughly equivalent to Java code (see also section 7.5). Straight integer calculation code is significantly slower than Java, as the compiler does

no unboxing[2] optimisations of any kind in its present state. Tests with various Java JIT compilers have shown decent speed and indicate Aardappel could be useful for everyday general purpose programming even in its present state.

6.4 The distributed runtime system

Besides providing infrastructure and utility functions for Aardappel code, the bulk of the runtime system has to do with concurrency, and in particular the distributed implementation of tree spaces. As the runtime system starts up, it tries to connect to tree space servers running on remote hosts (using RMI), either those specified on the command line or supplied by the environment. If it can connect to a host it will send the .class file needed to execute any code part of the program, which is then instantiated remotely. If no hosts are available it will simply perform all computation locally. Any new threads (resulting from new trees appearing in tree spaces) from the running program will transparently be distributed over the available hosts[3], and so will the location of new tuple spaces. Tuple space servers are separate programs that can be kept running on hosts, they are generic servers for any computation and can be "reused".

The runtime system implements the full Linda functionality, or rather the Aardappel superset of it: it supports obtaining multiple items atomically from the tree space. Furthermore, because Aardappel supports any number of tree spaces, the runtime system allows any of the tree space servers to host the actual tree space, so that communications overhead can be shared by all hosts (i.e. the client that started the computation does not necessarily become the bottleneck for a communication-bound program).

All of this is transparent to the compiled code as to which host it runs on, distributing data doesn't create any sharing problems (this is guaranteed at the language level), even built-in rules that involve state (such as graphics operations) are marshalled back to the originating system.

6.4.1 Performance

To get an idea of the performance (or lack thereof) of the tuple space runtime system, a simple `sum` program was run in different configurations. The sequential version simply

[2]Integers are represented as small objects, and for example an integer addition needs to take the two real integers out of the objects and construct a new integer object. Besides being slow in access, it creates a lot of garbage. Most professional functional language compilers optimize this away by static analysis.

[3]The scheduling method currently implemented is rather simplistic, but this can be easily be replaced, and Linda scheduling algorithms wasn't the focus of this research.

sums a list (1000 elements for all tests) and keeps track of the intermediate sum as a return value, i.e.:

```
sum([]) = 0
sum([h|t]) = h+sum(t)
```

To test tree space performance, a second version was used that stored the intermediate sum in the tree space, so does one `in` and one `out` operation per list element:

```
sum([]) = 0
sum([h|t]) { sumtotal(n) } = { sumtotal(n+h) } sum(t)
```

To test the difference pattern matching makes, a third version uses a tree with 3 extra dummy indirections, i.e. `sumtotal(a(b(c(n))))` instead of `sumtotal(n)`.

Benchmarking was done using the same setup described for sequential benchmarking in section 7.5. The second machine used in the LAN test was a Sun UltraSparc 300 running SunOS 5.6 and connected to the client machine over a 155Mbit local network. Results are as follows (all times are in seconds):

	single process	RMI localhost	RMI LAN
sequential	0.05	1.20	1.37
tree space	4.84	59.87	36.77
tree space + PM	4.88	62.12	39.17

Single process means the test was run within one JVM instance, so RMI wasn't used at all. RMI localhost has both the tree space server and the client program running on the same machine, and the LAN version has the client running separately on the aforementioned UltraSparc. Note that in all cases, the whole program consists of a single active thread (the recursive `sum`), which was forced to run remotely from the process that hosts the tree space (whenever applicable), so all tuple space operations have to go over RMI.

As is clear, the actual sequential overhead is virtually nonexistent, so we are purely testing tree space overhead. Slower times for the sequential sum over RMI are mostly connection overheads, as the whole sum is computed on one machine and only a single RMI exchange takes place (sending across a list of a 1000 elements).

Pattern matching overhead doesn't seem to be a factor either, as shown by the single process test. More pronounced overhead in the RMI runs is likely due to additional

marshalling overhead (each extra tree is an extra Java object that needs to be send across).

As expected, reading and writing to the tuple space on every element is a significant overhead over mere recursion, even in the single process case. But the big overhead is clearly RMI, and quite stunningly so if you realise that up about a minute is needed to perform 1000 times an `in` and an `out` operation. The reason that the LAN runs are actually faster than the localhost ones can be explained by the fact that the network used is very fast and most of the overhead is introduced by RMI, the workload of which is now shared over two machines[4]

6.5 Lessons learned

6.5.1 The Java VM as a backend

The Java VM is a good but not an ideal target. You get a lot for free:

- Portability, even for GUI/graphics.
- Having a garbage collector for a language that needs a garbage collector saves a lot of headaches.
- possibility of having code compiled but still retain interactivity thanks to Java's classloader system.
- ability to have editor, compiler, compiled code, runtime all in one system
- powerful persistency mechanism made loading/saving of code and distributing data through RMI straightforward.

The downside is that after using it for a while as language backend, it becomes painfully obvious that the Java VM (or at least a lot of VM implementations) wasn't written to support languages that differ greatly from Java in the stress they apply to various parts of the runtime, in particular:

- The garbage collector in most implementations is tuned towards small amounts of garbage being produced and objects that live relatively long, and thus can't cope with Aardappel's code that creates lots of objects that die young.

[4]To make sure these times were representative and not caused by different speeds of the machines, network stacks or JVMs, the LAN tests were rerun with the client and server roles reversed, which didn't result in any significantly different figures.

- Java Threads are relatively heavyweight, mostly using underlying operating system mechanisms to support it. Again threads in Java were clearly thought of as very coarse grain top level entities, whereas Aardappel code easily spawns large amounts of them. Most implementations can't cope with this and resort to crashing.
- The current compiler does no tail recursion optimisation so Aardappel's generally heavily recursive code made it run out of stack.
- RMI is a lot more heavyweight than one would expect, and again intended for coarse grain usage. Communication intensive Aardappel code wouldn't crash it, but still make it slow down severely.

It is these items combined that hold Aardappel's implementation back for practical use: what good is a language with advanced features if you have to restrict your usage of them because the underlying system will choke if you use too much of it?

6.5.2 Graphical programming

Using the system for actual programming has been a very valuable and interesting experience, and has created a lot of insight into the value of the design choices made in Aardappel and graphical programming in general. As suggested earlier, it is still hard to compete with textual syntax. You can't really win against textual identifiers, but you can get close, and experience with Aardappel in a practical context has shown that example values have been the most successful contestant yet. In all the code written, using examples instead of identifiers as placeholders has sometimes been superior (when there is a clear structure to the data) and sometimes been inferior (for example with integers). On the whole it is a big win over other methods of graphical programming as it allows one to track relationships and their meaning easier than for example in dataflow languages.

Another very positive experience was being able to do code editing primarily by drag and drop, and the smart auto-layout of the tree structure. Editing in general can be improved from the current version, there are certain operations which are disproportionately hard compared to others (for example wrapping a tree with another tree as opposed to editing its children), and other operations weren't consistent enough (some combinations of dragging/replacing trees could create unwanted sharing or lose a sharing relationship). The good news is that improving on the editing/layout mechanisms is completely independent from the language and compiler, if a new technique is thought of it can be implemented and all code, old and new, can immediately profit from it.

6.6 Future work

There is a lot of work that can be done on the implementation. As mentioned above, from a graphical programming point of view a lot can be gained by smarter editing/layout mechanisms to complement the existing ones. From the perspective of a practical general purpose programming language it needs a more complete set of built-in functions, in particular some well designed interfaces to the Java API's. But the most interesting future work from a research perspective is code optimisation and analysis specific to the Aardappel semantics. Some possible directions are outlined below.

6.6.1 Soft type inference

The type system of the Aardappel Language is purely dynamic. It is not an issue whether a language like this should have static/strong or dynamic typing: the dynamic nature of tree spaces coupled with the semantics of Aardappel's version of tree rewriting make a static type system impossible or at best very artificial and cumbersome. Rather, the question is how to best go about supporting it.

Dynamic typing, contrary to popular belief, does not need to imply loss of static information on the correct application of arguments to values, nor the loss of opportunities to optimize the code. There's nothing against having a type inferencer in the *implementation* (rather than the language), that can give you advice (rather than blunt errors) about possible problems with your code (see e.g. [2] [1]).

This can give you the best of both worlds: the language will keep its semantic simplicity, purity and the beauty that dynamic typing gives you and won't be arbitrarily restricted. The type inferencer can be more powerful than one present in a language as its semantics is hidden from the user, it can even improve from implementation to implementation as the understanding of either the language semantics or type systems in general progresses (the implementation will simply be more and more precise about its "advice"). A type inferencer in an implementation will thus try to make an as precise as possible *conservative* estimate about the correctness of the program regarding types, i.e. the set `A` of programs it accepts is a proper superset of the set `B` of programs that will run without runtime type errors, whereas the set `C` that a static typing system accepts is an arbitrary subset of `B`, based on the current state of the art in type systems at that point in time, or the whim of the designer. The proper subset of `A` which is equal to or larger than `C` will be accepted without complaints, whereas `A-C` (the rest, with bits on both sides of the `B` boundary) is the set it will give advice about, as precise as its technology will allow.

6.6.2 Optimisation

Type inference and other analysis (as outlined in the previous section) will give an optimizer a lot of information to play with to optimize at least some algorithms to the speed of today's imperative languages (or better, as it has no aliasing problems to contend with). Yet there are some particular features of Aardappel that specifically need some degree of analysis and optimisation if significant slowdown is to be avoided.

Aardappel's rule selection mechanism has a worst case of dynamically having to find the appropriate rule based on a number of arguments. The average case is often a lot simpler, and with a basic analysis a fairly optimal hardcoded dispatch can be determined. Clever specialization and inlining can help here.

Another area identified above as being a great slowdown is garbage collection. Aardappel was implemented using the garbage collection of the underlying system because that was the easiest thing to do, but Aardappel's semantics make a linear implementation particularly attractive.

The feature with the poorest worst case behaviour though is the Linda "in" operation. Ideally we want programmers to be able to access the tree space as an O(1) operation, as that would give them most freedom in structuring their programs, but the simplest implementation of "in" is O(n) worst case, and even though the current implementation will give you O(1) as the average case and there are many more optimisations imaginable, no analysis I've been able to think of is going to prevent a worst case altogether. The Yale Linda group haven't been able to overcome this problem either, but then again they can't potentially do an as precise analysis (their language is C, and used in an open-world situation [15]). Clearly novel analysis or otherwise profiling techniques are called for.

Another easy performance win compared to the current implementation is to improve the thread distribution algorithm, using more sophisticated scheduling.

Chapter 7

A case study

This chapter presents a small case study of a program written in three languages (Aardappel, Java and Haskell [36]), to point out some differences in the way they can share data, and their programming style in general.

7.1 Problem description

The system implemented in all three languages is that of a simplistic airplane booking system[1]. It was chosen because it deals mainly with processing and access of data, and nothing else. The implementations should model the following:

- A set of clients that want to book flights from somewhere to a destination, preferably on a certain day.
- A set of agencies that (concurrently when possible) service clients and use a central booking service to find the best possible flight(s) for the client. The agent should try, in this order:
  - A direct flight.
  - A flight with one stop.
  - One of the two above on the next day, until a flight is found.
- A central booking service that administers the flights, seats free on them and seats already booked, and allows at most one agency at a time to make a booking.

[1]Though a familiar example, it is not an existing or classic example, rather something I constructed myself to suit the needs of the discussion.

No error checking needs to be done, i.e. it is assumed clients don't specify a bogus destination, or there won't be a plane at a later date to satisfy a request etc.

To see how the different languages deal with program extension, we specify a small enhancement to be made that manipulates the state: any flights that are fully booked should be put aside from the set of flights (to speed up the search).

7.2 The Aardappel version

Both a textual and graphical version are given, but we will comment on the code by means of the textual version first:

```
{
  agency(0)
  agency(0)
  agency(0)

  client(dave,london,rome,1)
  client(harry,london,paris,1)
  client(john,london,berlin,1)

  flights({
    plane(london,paris,1,[23,45,3],[])
    plane(paris,rome,1,[83,58],[])
    plane(london,berlin,1,[67],[])
  })
}
```

This is the main tree space, where agents, clients, and the central booking system are all modelled as (concurrent) trees. Agencies count the number of clients they have serviced (just to give them some state). Clients specify their name, source and destination for their trip, and preferred date. A date is modelled abstractly as a number, where higher numbers mean a later date.

We model the available flights as a local tree space. In most Aardappel programs it would have been just as convenient to put everything in the main tree space, but then it would have been more difficult to control access to them and ensure one agency can't grab that vital last seat of a complex booking before it has been completed. Local tree

spaces are all about structuring state and making things easier to overview and verify their correct behaviour, and that is what we are doing here.

A flight (the `plane` trees) holds (besides the source, destination and date) two lists, the first of which is the available free seats, and the second the names of the people that have already booked a seat (here just given as empty so far, to not clutter the example).

```
agency(n) { client(name,a,b,date) flights(b) } = { flights(nb) }
  agency(n+1) when nb = { book(name,a,b,date) flatten(b) }
```

An agency repeatedly tries to grab both a client, and access to the central booking system. If it manages to grab both, it generates a booking process (the `book` tree) which is put in a separate tree space with all the flights for the purpose of making the booking (`flatten` is a built-in function that flattens tree spaces, but unlike the familiar list function it does this from within, i.e. it adds all elements of `b` to the surrounding tree space). The purpose of this exercise is of course to make things easy on ourselves: we know that the state (the flights) can only be acted upon by the booking process, until it's done. This trivialises the possible scenarios.

We use a local binding (using `when`) to evaluate the local tree space to artificially enforce that the agency can't run off and find a next client until the booking has completed (for the sake of realism).

```
book(name,a,b,date) { plane(a,b,date,[seat|seats],people) } =
  plane(a,b,date,seats,[name|people])

book(name,a,b,date) { plane(a,c,date,[seat|seats],people)
                      plane(c,b,date,[seat2|seats2],people2) } =
  { plane(a,c,date,seats,[name|people]) }
    plane(c,b,date,seats2,[name|people2])

book(name,a,b,date) = book(name,a,b,date+1)
```

Finally, the actual booking process. These three rules have the same specificity, so they will be tried in this order. The first rule models the direct flight case: if there is a flight on that date with the right source and destination, then we are done booking and the result is that flight, with one seat removed and one name added to the list of passengers.

Note the use of a lesser known feature of Aardappel: pattern matching on values through repeated variable occurrences. Most functional languages with pattern matching[2] choose not to allow this, because it often doesn't bring a lot of benefit, but in Aardappel this feature is crucial as the Linda `in` operation is bound to left hand sides of rules, and if it weren't for this feature, there would be no way to grab a specific tree from the tree space according to a dynamic value.

The second rule is more interesting, as it used the repeated variable feature in a more powerful way: it grabs two flights where the destination of the first is the source of the second, and the rest matches the clients trip. This is a relatively expensive operation, but allows for very concise notation, reminiscent of logic programming, because finding the two flights requires an exhaustive search[3] similar to backtracking over data organized by `assert` / `retract` or a blackboard, with code that would look very similar.

The third rule simply bumps the day by 1 and continues searching.

Should this search succeed, then the booking is again done and the result is both `plane` trees with updated seat and passenger information. Another Aardappel peculiarity shown here is that if a Linda `out` happens on the top level of a rule and also the tree being reduced wasn't part of another tree (i.e. sits directly in the tree space itself) it doesn't really matter which of the two trees is put into the tree space and which is the result of the whole, they could easily have been interchanged.

```
plane(a,b,date,[],people) = full_plane(a,b,date,people)
```

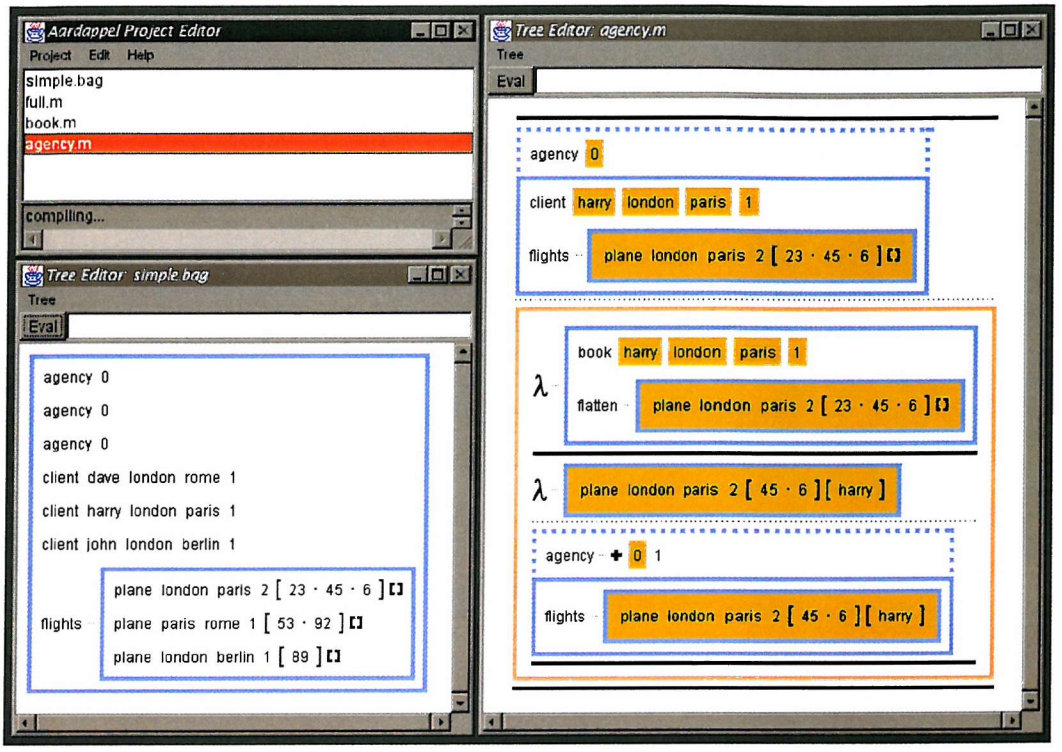
This is the single rule we add to implement the added functionality of filtering out fully booked flights, if slightly crude. Notice another property here which is hard to replicate in non-tree rewriting languages: even though a plane is considered passive data, the language makes no such distinction. Rules like this allow one to perform checks on certain variations and change them accordingly, without influencing its role as a data structure. Thanks to the way rewriting of rules works (i.e. if you pattern match on a child it is guaranteed to be in normal form) and the fact that the language doesn't support in-place mutation, this rule is guaranteed to be applied on any new or modified `plane` tree before any other rule can access it, so it is easy to guarantee certain invariants. It is as if this check gets inserted automatically everywhere in the program

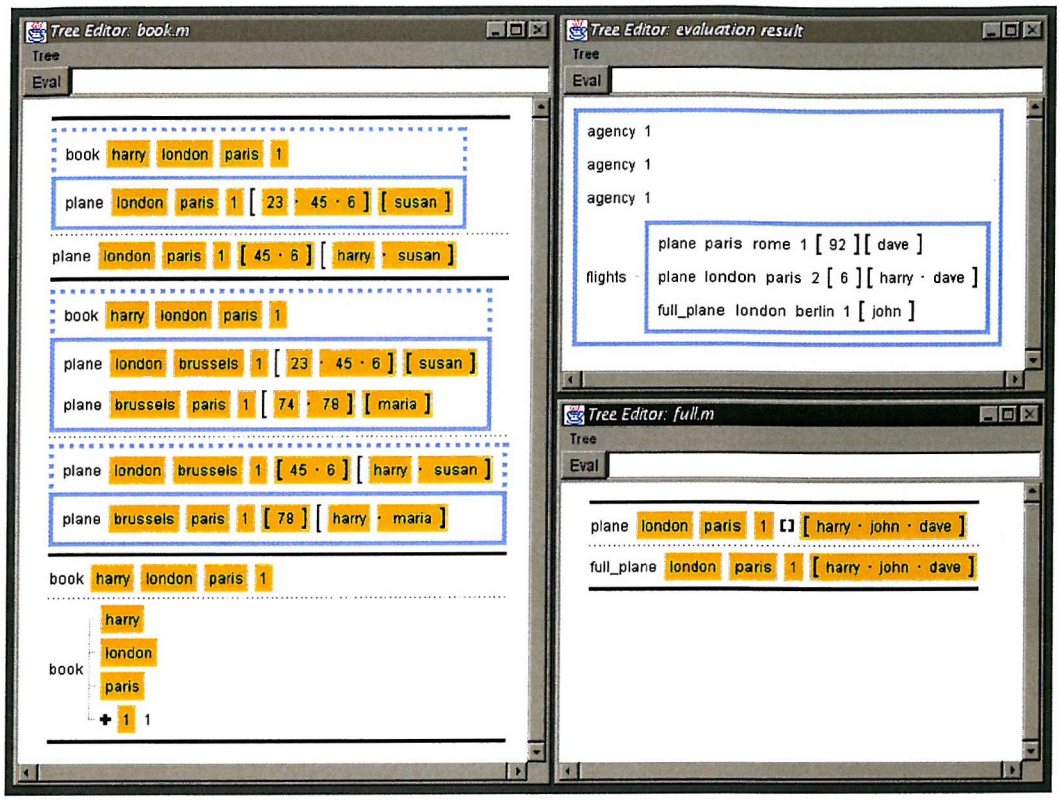
[2]Logical languages however use this extensively, as it is part of unification by definition (dynamically, because of first class variables).

[3]Even though this has always been part of the design, and specified correctly in the formal semantics, the optimisations of tuple space operations in the implementation were wrong and partially broke this behaviour, which I found out the hard way when testing this code.

where needed. A functional language could model this behaviour, but not transparently (and not without planning ahead).

Below is the graphical version, which is identical in structure to the textual version above, except for the `when` construct which has been replaced by a local rule.





Recall that examples (place holders) are marked by an orange background. Some example values such as the one representing the bag of flights in agency rule are quite big, even though they only contain one flight, but it is important to use example values that are possible sample / test cases for the code, as this allows immediate testing of sub-expressions by selecting them and pressing eval (see section 3.4). Note how the example values, though maybe not easily readable on first sight, give a good structure to the code. The top right window above shows the output (not shown in the textual version).

7.3 The Java version

Java is a good representative of all imperative and object oriented languages. Even though it is a descendant of C, it is a very clean and well designed language, and comes with good abstraction facilities to model any system. Because Java has threads as part of the language, we'll use them as part of our implementation to model concurrent behaviour of agencies to make things a little bit more interesting.

```
import java.util.*;
```

```
class PlaneReservation {

  static void main(String args[]) {

     ClientQueue cq = new ClientQueue();
     cq.put(new Client("dave","london","rome",1));
     cq.put(new Client("harry","london","paris",1));
     cq.put(new Client("john","london","berlin",1));

     Flights f = new Flights();
     Plane p;
     f.put(p = new Plane("london","paris",1));
     p.freeseat(63);
     p.freeseat(27);
     p.freeseat(87);
     f.put(p = new Plane("paris","rome",1));
     p.freeseat(55);
     p.freeseat(12);
     f.put(p = new Plane("london","berlin",1));
     p.freeseat(55);

     Agency a1 = new Agency(cq,f);
     Agency a2 = new Agency(cq,f);
     Agency a3 = new Agency(cq,f);
     a1.waitForTermination();
     a2.waitForTermination();
     a3.waitForTermination();

     f.report();
     System.exit(0);

  }

}
```

In the main program, in analogy to the main tree space, we create all objects and set them going. In Java we have to be more explicit about the roles of all elements in our

booking, and we chose to model all state (the central booking system, but also the list of clients) as monitors (objects with semaphore facilities), and agencies as threads accessing them.

```
class ClientQueue {
  Vector q = new Vector();

  synchronized void put(Client c) {
    q.addElement(c);
    notify();
  }

  synchronized Client get() {
    while(q.size()==0) {
      try { wait(); } catch(InterruptedException e) {};
    };
    Client c = (Client)q.elementAt(0);
    q.removeElementAt(0);
    return c;
  }

  synchronized boolean empty() {
    return q.size()==0;
  }
}
```

The `ClientQueue` is a very basic monitor: it allows us to store and retrieve objects, but only one thread at a time. `synchronized` tells Java to put a caller (a thread) in a wait queue if the object is already in use. As soon as a caller enters a method it locks the object, and releases it when it leaves the method. Should it enter the `get` method when there are no clients in the queue (yet), it will be put to sleep and the object will be unlocked. If another thread adds a `Client` object in `put`, `notify()` will cause any waiting threads to be woken up again (by throwing `InterruptedException`) so it can try again to grab an element.

Monitors make a concurrent program safe. This is a necessity, as the problems imperative programs cause when allowing multiple parts of a program to mutate / look at a shared memory location grow proportionally worse if multiple threads are allowed access

at the same time. In our example, the queue has been made more robust, we could even add the clients after we start the agencies, and the behaviour of the program wouldn't change.

The empty method is there for callers who want to know if there are any more clients without the risk of being blocked.

```
class Client {
  String name, from, to;
  int date;

  Client(String n, String f, String t, int d) {
    name = n; from = f; to = t; date = d;
  }
}
```

A minimal class to define a new datatype. We haven't defined any accessor methods, we will access the instance variables directly to keep the example small.

```
class Plane {
  String from, to;
  int date;
  Vector seats = new Vector();
  Vector people = new Vector();

  Plane(String f, String t, int d) {
    from = f; to = t; date = d;
  }

  void freeseat(int n) {
    seats.addElement(new Integer(n));
  }

  void report() {
    System.out.print("plane from "+from+" to "+
      to+" on "+date+" passengers ( ");
    for(int i = 0;i<people.size();i++) {
      System.out.print(people.elementAt(i)+" ");
```

```
    };
    System.out.print(") free seats ( ");
    for(int i = 0;i<seats.size();i++) {
      System.out.print(seats.elementAt(i)+" ");
    };
    System.out.println(")");
  }
}
```

Plane is another simple datatype. Lists in Java are best modelled using `Vector`, which is a more convenient version of an array (which grows automatically). We added a report method to be able to see the final result of all bookings.

```
class Flights {
  Vector ps = new Vector();

  synchronized void put(Plane p) {
    ps.addElement(p);
    notify();
  }

  synchronized void book(String name, String from, String to,
                                                    int date) {
    while(ps.size()==0) {
      try { wait(); } catch(InterruptedException e) {};
    };

    for(;;) {

      for(int i = 0;i<ps.size();i++) {
        Plane p = (Plane)ps.elementAt(i);
          if(p.from.compareTo(from)==0 &&
             p.to.compareTo(to)==0 &&
             p.date==date &&
             p.seats.size()!=0) {
               p.seats.removeElementAt(0);
               p.people.addElement(name);
               return;
```

```
        };
      };

      for(int i = 0;i<ps.size();i++) {
        Plane p = (Plane)ps.elementAt(i);
        for(int j = 0;j<ps.size();j++) {
          Plane s = (Plane)ps.elementAt(j);
          if(j!=i &&
             p.from.compareTo(from)==0 &&
             s.to.compareTo(to)==0 &&
             p.to.compareTo(s.from)==0 &&
             p.date==date &&
             s.date==date &&
             p.seats.size()!=0 &&
             s.seats.size()!=0) {
               p.seats.removeElementAt(0);
               p.people.addElement(name);
               s.seats.removeElementAt(0);
               s.people.addElement(name);
               return;
          };
        };
      };

      date++;

    };

}

synchronized void report() {
  for(int i = 0;i<ps.size();i++) {
    Plane p = (Plane)ps.elementAt(i);
    p.report();
  };
}
```

```
}
```

`Flights` models the central booking system, and this is where all the actual searching happens. We do this in classic imperative fashion, two loops attempting to do a direct flight and a one stop flight respectively. In the latter, we simply try out all possible combinations. If we find a suitable booking we reserve it by grabbing a seat and adding the person to the flights in question, and if we fail we try the same thing again the next day.

```
class Agency extends Thread {
  ClientQueue cqueue;
  Flights flights;
  boolean done = false;

  Agency(ClientQueue cq, Flights f) {
    cqueue = cq;
    flights = f;
    start();
  }

  public synchronized void run() {
    while(!cqueue.empty()) {
      Client c = cqueue.get();
      flights.book(c.name, c.from, c.to, c.date);
    };
    done = true;
    notifyAll();
  }

  synchronized void waitForTermination() {
    while(!done) {
      try { wait(); } catch(InterruptedException e) {};
    };
  }

}
```

`Agency` is the active party in this program. To make it a thread all we have to is inherit from `Thread`, and call `start` on the object (which we do in the constructor). Once Java has created the new thread it will call `run`, where we try and grab clients until there are no more. The main program creates 3 of these objects. `Agency` is also its own monitor, for the sole purpose of allowing the main thread to wait on its termination, to terminate the program correctly.

When run, the program generates the following output:

```
plane from london to paris on 1 passengers ( dave harry ) free seats ( 87 )
plane from paris to rome on 1 passengers ( dave ) free seats ( 12 )
plane from london to berlin on 1 passengers ( john ) free seats ( )
```

Comparing the Java version to the Aardappel program it is obvious the Java program is much bigger, and contains much more "unnecessary" code, but this is not fair to Java. Java is a more low level language and as a consequence more verbose and requires doing a lot of things manually that come for free in Aardappel (most prominently searching), but it is possible that if the programs were bigger and / or more numerical in nature the difference wouldn't be as big.

Programming with monitors in Java and writing well designed and properly encapsulated objects is about the closest you get in an imperative language to the airtight state and concurrency encapsulation of Aardappel. The downside is that solid design is required for it to function optimally, anything less will introduce classes that are not really robust (often hard to spot until a certain condition arises because of non-determinism) with all the familiar consequences. As an example, just glancing over the code again I spotted a small bug: in the `while` loop of `Agency.run` it is possible for another agency to grab the last client just after the current thread has determined there are still clients left, causing it to block, and maybe as a consequence the program not to terminate.

This is just a small program, but it suffices to say that a language like Java makes no techniques available to guarantee that a certain design is protected against problems like this, or general problems originating from shared access.

Next, we try to extend the Java program with the functionality of setting aside fully booked flights. The easiest way was to add a new list of full flights to `Flights`:

```
Vector full = new Vector();
```

Then create a new method to check for a full flight, and move to `full` vector if necessary:

```
void checkfull(int i, Plane p) {
  if(p.seats.size()==0) {
    full.addElement(p);
    ps.removeElementAt(i);
  };
}
```

We call this function from where ever a `Plane` is modified, i.e. in the method book in the first loop just before `return`:

```
checkfull(i,p);
```

And again in the second:

```
checkfull(i,p);
checkfull(j,s);
```

Also, we have to modify the `report` function to show full flights as well (refactored into 2 methods, the one below replaces the original which then gets abstracted using an extra argument):

```
synchronized void report() {
  report(ps);
  System.out.println(''full planes:'');
  report(full);
}
```

As you can expect from an imperative language, it is very easy to do, but it also relies heavily on the amount of places that access state (the dependencies), which in our case was very limited. More importantly, there is no way to guarantee the modification has covered all possible dependencies (as you can in Aardappel), which can cause bugs which are difficult to discover and fix. Extension in OO is supposed to be done using inheritance, but in this case that wasn't possible.

7.4 The Haskell version

Haskell is as a language much closer to Aardappel than Java, but in the case of the problem at hand the languages offer very different facilities. Haskell doesn't really have concurrency as part of the language (even though many implementations and dialects do), so the program was made purely sequentially, with agencies as seperate entities completely omitted and replaced by a sequential loop processing all clients.

In Haskell there are basically two options of doing state: good old-fashioned functional programming (i.e. lots of plumbing) or using Monads. I decided it was most appropriate to use the latter, but ended up doing a bit of both.

```
data Client = CL String String String Int
data Plane = PL String String Int [Int] [String]

type BState = ([Client],[Plane])
```

These are the central data types we use. A `Client` and a `Plane`, much like in Aardappel and Java, and the overall state of our program (`BState`), which is a list of clients and a list of flights.

```
newtype BMonad a = BM (BState -> (a,BState))

instance Monad BMonad where
  return x = BM (\s -> (x,s))
  (BM a) >>= b = BM (\s -> let (x,s2) = a s
                               (BM q) = b x
                           in q s2)
```

Here we define our monad, `BMonad`. Monads are a very clever functional programming idiom (as opposed to a language feature), that was designed to hide plumbing of all values representing the state in a computation[4]. It does this by requiring all composition of computation (i.e. calling a function on the result of another, etc.) to use special functions instead that are able to plumb the state behind the scenes. Needless to say, replacing functionality so basic at the language level makes writing code very cumbersome, but Haskell gets around this using some heavy syntactic sugar.

[4]It can do many other things besides, such as model exceptions, backtracking, I/O etc., but that is not our concern here.

Our monad is an instance of the `Monad` class defined in the Haskell prelude (a class in Haskell is very similar to what is denoted as a "type" in Aardappel in section 3.3.2 and 4.1.1, i.e. a set of operations supported by a certain datatype, and very unlike Java, i.e. a template of an object). A monad value (think of it as a "stateful" value) is modelled as a function from one state to another plus the actual value. They are functions (closures) to facilitate easy chaining of computations, and wrapping them in a datatype is because Haskell requires instances to be datatypes.

Being an instance means we only have to define the operations that are different for `BMonad` (`return` and `>>=`), as all other monad functions are defined in terms of these two (again, similar functionality to Aardappel's polymorphism). `return` converts any value to a monad value, and `>>=` implements sequencing of function calls and passing on their results[5]. For example, an expression previously written `f(g(1))` now has to be written `return 1 >>= (\x -> g(x) >>= (\y -> f(y)))`, but the advantage is that if f and g were carrying a lot of state around it could now be hidden in the monad value, easily extended with new state, and any part of that expression can access the state without disturbing the rest of the expression.

```
getCL = BM(\(x,y)-> (x,(x,y)))
getPL = BM(\(x,y)-> (y,(x,y)))

putCL x' = BM(\(x,y)->((),(x',y)))
putPL y' = BM(\(x,y)->((),(x,y')))

applyBM (BM p) s = p s
```

These functions are there to conveniently access and modify the state present in a monad value: they pattern match on the internal state of a monad, and either copy the value out or replace with a new value. They are for the list of `Client` and `Plane` values, respectively. `applyBM` is a special version of normal function application that deals with our monad as function value.

```
clients = [(CL ''dave'' ''london'' ''rome'' 1),
           (CL "harry" "london" "paris" 1),
           (CL "john" "london" "berlin" 1)]

planes = [(PL "london" "paris" 1 [54,24,6] []),
```

[5]It is called bind in many earlier papers on monads.

```
          (PL "paris" "rome" 1 [55,23] []),
          (PL "london" "berlin" 1 [35] [])]

main = fst (applyBM dobookings (clients,planes))
```

This calls the main booking loop. The result value is the first element of the state tuple, which is the actual return value (not the state).

```
dobookings = do cs <- getCL
                ps <- getPL
                (if (null cs)
                 then
                     return ps
                 else
                     let (cl:rest) = cs in
                       do putCL rest
                          putPL (booking cl ps)
                          dobookings)
```

The main loop, which looks quite imperative with all the syntactic sugar in place. It uses "assignments" and has hidden state, and even though it is in theory still referentially transparent it is not as easy to read anymore. The do syntax is short for a sequence of >>= operators and lambdas. It starts by getting both lists of clients and flights from the monad, and if the client list is empty we return the flights as result of the function. If not, we grab a client and "assign" the remainder back to the state, and call the booking function to perform the actual booking. The result (with potentially modified flights) gets assigned back to the state as well, and we call the function recursively.

```
booking (CL name from to date) ps = book [] ps where

  book _ [] = bookvia [] ps
  book a (pl@(PL f t d seats people):rest) =
    if (not (null seats)) && from==f && to==t && date==d
    then a ++ ((PL f t d (tail seats) (name:people)):rest)
    else book (pl:a) rest

  bookvia _ [] = booking (CL name from to (date+1)) ps
  bookvia a (pl@(PL f t d seats people):rest) =
```

```
    if (not (null seats)) && from==f && date==d && (not (null second))
    then second
    else bookvia (pl:a) rest
      where second = booksecond [] (PL f t d (tail seats)
                                    (name:people)) t (rest ++ a)

booksecond _ first from2 [] = []
booksecond a first from2 ((pl@(PL f t d seats people)):rest) =
  if (not (null seats)) && from2==f && to==t && date==d
  then a ++ first:((PL f t d (tail seats) (name:people)):rest)
  else booksecond (pl:a) first from2 rest
```

We could have programmed the function booking in monad style as well, but with a not entirely trivial piece of code like this that would have created a very large and not very readable piece of code. Instead, this core function was written using regular plumbing style, a bit reduced by using local function definitions, so the read only arguments (the client requirements) can be accessed as free variables. book tries to find a direct flight, and upon failure passes on control to bookvia, which attempts to find a one stop flight. booksecond is used by bookvia to find the second leg of a trip. Similar to the Aardappel code the second leg only has access to the remaining flights, but here we have to code it ourselves. If bookvia fails, it recurses into booking again, one day later.

The output looks very predictable and familiar:

```
[Plane_PL ''paris'' ''rome'' 1 [23] [''dave''],
 Plane_PL ''london'' ''paris'' 1 [6] [''harry'',''dave''],
 Plane_PL ''london'' ''berlin'' 1 [] [''john'']] :: [Plane]
```

Extending the program with extra functionality to remove fully booked flights is relatively the most difficult of the three languages, as expected. Adding the code at the place where the actual modification happens (in the various local functions of booking) would make a mess of things, as it requires just about every line of code to be changed to pass the new list around. To circumvent this we will do it at monad level where we can add new state, and simply filter the full flights out of the result of booking:

```
dobookings =
  do cs <- getCL
```

```
ps <- getPL
full <- getFL
(if (null cs)
 then
     return (ps,''full:'',full)
 else
     let (cl:rest) = cs
         b = (booking cl ps)
     in
       do putCL rest
          putPL (filter (\(PL from to date seats people)
                          -> not (null seats)) b)
          putFL (full ++ (filter (\(PL from to date seats people)
                                   -> null seats) b))
          dobookings)
```

To store the second list of planes we add a new bit of state to our monad:

```
type BState = ([Client],[Plane],[Plane])
```

And we to access it we modify and extend our accessor functions:

```
getCL = BM(\(x,y,z)-> (x,(x,y,z)))
getPL = BM(\(x,y,z)-> (y,(x,y,z)))
getFL = BM(\(x,y,z)-> (z,(x,y,z)))

putCL x' = BM(\(x,y,z)->((),(x',y,z)))
putPL y' = BM(\(x,y,z)->((),(x,y',z)))
putFL z' = BM(\(x,y,z)->((),(x,y,z')))
```

Monads solve the problem of adding extra state quite well, and make it almost as easy as in an imperative language. However with that comes the same opacity of imperative languages. The biggest stumbling block however is that monad code, though looking relatively straightforward in these examples, is not easy to write, especially constructing more complex monads themselves, and their actual usage intermingled with functional code (to such an extent that I regard it almost an impossibility that an "average" programmer can write a complex program with monads).

Regular functional code (as exemplified by the booking function) result in concise and precise code, that makes all dependencies explicit and is easy to read, but also makes changing access to state a lot of work.

7.5 Performance comparison

The previous sections compared Aardappel to Java and Haskell in terms of code size, clarity, and ease of modification, for each of which we can tentatively say that Aardappel scored best, Haskell came second, and Java performed relatively poorest. Though not a focus of this thesis, in this section we want to give the reader an impression of Aardappel's relative performance in terms of execution speed and memory usage.

For benchmarking we chose to use a simple quicksort implementation, because it is easy to scale up to heavy work loads, performs all typical operations such as function calling / dispatch, data structure construction and selection, etc.[6], and most of all is easy to ensure that code is equal across all languages (equal meaning it has similar functionality and is guaranteed to have the same workload, i.e. exactly the same data was processed).

The benchmarking setup was as follows:

- Aardappel and Java code were both run on the Sun JDK 1.2, Haskell code was run from HUGS[40]. A Haskell compiler that compiles to the JVM (for a more even comparison) would have been preferable, and one is under development [68], but not publicly available at this time (cited times for the current version typically are 2 to 8 times as slow as HUGS, so this would unlikely have improved the situation for Haskell). Faster Haskell compilers do exist, but not available to the author at the time of writing.
- Times were measured for quicksorting lists of random numbers of sizes 200, 1000 and 5000[7]. Each benchmark consists of 100 of these `qsort` calls in sequence, to improve timing accuracy and amortise garbage collection cost over all runs.
- All benchmarks were run at least 3 times and the best time taken (to make sure all relevant code is swapped in).

[6] We avoided a purely numerical benchmark, as that would become merely a test of each compiler's unboxing optimisations. Similarly a benchmark involving concurrency (such as the flight booking example) was avoided, because not all implementations support concurrency, or at least vary widely. For notes on the efficiency of Aardappel's concurrency mechanism see section 6.4.

[7] This upper bound was chosen because above it HUGS was running out of stack space (and the JVM just above that).

- Time measurements were obtained using Java's `System.currentTimeMillis()` function for Aardappel and Java and as such were precise given the run time of the benchmarks (though variance was existent, even on repeated runs). Haskell code was timed using a software stopwatch, giving accuracy to within half a second.
- Benchmarking was performed on a Pentium pro 200 with 128 megabyte of ram, running the Windows 95 operating system and only a few idle applications running.
- Great care was taken towards code equivalency across the implementation, for example all three implementations use the same custom random number generator to ensure workload for quicksort is identical. All three versions also use non-mutable list code, but since this is not natural for Java, a seperate in-place update version was also tested.

The results obtained are given in the following table, which shows execution time in seconds for 100 sorts of lists of different lengths, for each of the languages:

	200	1000	5000
Aardappel	1.37	12.41	92.30
Java	1.32	10.22	97.20
Java (in-place update)	1.37	10.54	110.40
Haskell	47.00	325.00	3200.00

For this benchmark at least, Aardappel is able to keep up well with Java. Compiled Aardappel code is (in its current stage) less optimised than the equivalent Java code and often constructs more objects than handwritten Java, but one of its big advantages above Java is that it compiles all code to static methods (essentially functions) which saves a significant amount compared to the expensive polymorphic dispatch Java does by default (which in Aardappel is replaced by pattern matching code).

Using in-place update does not seem to make much of a difference, which can be attributed to the fact what is saved on new object allocations is lost on extra selector and mutator method calls. There are a lot of other parameters that could have been tweaked about the Java code (i.e. do away with data hiding and access instance variables directly, or specialise the code to be hardcoded for integers), but that would not have made for a fair comparison.

Haskell loses out by large amounts, which is primarily due to HUGS being a simple interpreter which doesn't do a lot of optimisations, which in turn probably pronounces the overhead associated with lazy evaluation more than necessary (closure creation etc.).

Memory profiling was done using a Java memory profiling tool, which allowed us to measure the live data (maximum amount of in-use allocated objects at any one time, in kilobytes), program data ratio (percentage of live data that belongs to the running program, the rest being data allocated by the JVM, such as class data), and the total memory footprint (maximum allocated memory including garbage objects at any one time, in kilobytes). The program is the same quicksort test above, for a list length of 1000:

	live data	program data	footprint
Aardappel	600	70%	1600
Java	500	50%	1500
Java (in-place update)	400	15%	500
Haskell	-	-	-

The garbage collector for the Sun JVM would be invoked after 1 megabyte of new objects were allocated, irrespective of the maximum heap size, so the memory footprint would always be at most 1 megabyte more than the live data (and less if the heap size were shrunk manually).

As shown, Aardappel allocates more objects than the Java version. The Java in-place update version hardly allocates more objects than minimally needed, and doesn't provoke a garbage collection at all.

It wasn't however possible to obtain similar statistics for Haskell, all information supplied by HUGS is the total number of cells allocated during the process of evaluation, but since HUGS performs very frequent garbage collections, this gives us no estimate as to the live data at any one time (similarly, it wasn't possible to obtain the total number of allocations from the JVM). One can estimate that per unit of work, HUGS allocates significantly more memory than Aardappel (for closures), but given the code similarity it is still likely to be within the same order of magnitude.

As a final metric, the .class file code sizes for Aardappel and Java are very similar: about 5 and 3 kilobytes respectively.

While the test do not constitute a comprehensive comparison, they do suggest that the Aardappel implementation generates code that has similar efficiency to that produced by the Java compiler.

Chapter 8

Related work

This chapter points to research that has been done which relates to Aardappel or upon which Aardappel is based. Where appropriate the alternative approach is discussed.

8.1 Tree rewriting

An area where the computational model of tree rewriting has been very popular is semantic specification. Its expressiveness in dealing with complex data structures makes a specification very short and readable, and its declarativeness, simplicity and intuitiveness makes sure the semantics of the specification language never obscures the semantics of the language actually being specified (it makes it very "invisible"). Many formalisms using something akin to tree (term) rewriting exist (e.g. ASF+SDF [34]), but they are not within the focus of this research.

Besides its great popularity in specification languages and calculi, Tree rewriting as a basis for a programming language (as used in this research) is an almost unexplored field, even though it is similar to functional programming. One of the earliest and best known attempts in this area is by O'Donnell [51] who extensively looks at evaluation mechanisms, tricks and optimisations for a language based on a form of tree rewriting (which he calls "equational logic"). Subsequent progress for this type of language has largely been in the area of optimization of compiling rules and pattern matching (e.g. [60] [61]). Other programming languages based on tree rewriting are Q [31], Hielp [5], and ELP, which is an implementation of O'Donnell's system. All of these are pretty bare implementations of the basic tree rewriting semantics, all with slightly different choices for evaluation tactics as discussed in section 2.1, but none use features which

fundamentally extend the expressive power of tree rewriting such as an outermost-outermost evaluation strategy or rule selection strategies like specificity ordering, and therefore do not contribute anything interesting from an Aardappel perspective. Most recently XSLT[20] (a language for translating XML [11] documents into other XML documents) is based on the tree rewriting mechanism.

Probably the biggest area of development of languages that use rewriting as a programming language mechanism has been graphical languages (see section below). From an evaluation point of view they use something that has a worse complexity than either innermost or outermost evaluation because graphs / grids and other visual structures don't have an intrinsic logical search order embedded in them.

The system of inclusion polymorphism that results from Aardappel's specificity ordering has similarities to multiple inheritance of interfaces as found in many modern OO languages (as explained in section 4.1). It is also similar to Poly [47] which allows you to specify an interface on the fly as a function argument by merely specifying a function that a type needs to have implemented to be a valid argument. To some extent it is similar to Haskell type classes [10] [55] [41], which also organize types from the perspective of "functions implemented for them", but not on the fly as in Poly and Aardappel.

In OO languages, dispatching on multiple arguments like Aardappel does is called *MultiMethods* (used in languages like CLOS [27] and Cecil[18]), but as mentioned before, the dispatching mechanism of MultiMethods is slightly more involved as it correctly dispatches subclasses as well.

8.2 Linda

The idea of a tuple space was invented by Carriero and Gelernter in their language Linda [17] [13] [28] [16] [14]. The most well known language to implement the Linda model is C-Linda [49] (see section 4.2 for a comparison between C-Linda semantics and Aardappel).

There are a great number of coordination languages that extend the basic Linda model, for example to include multiple (first class) tuple spaces like Bauhaus Linda [12] or Bonita [57]. Bauhaus Linda merges the concept of a tuple and a tuple space, and allows them to be used as values and be arbitrarily nested. Furthermore, it represents processes as values in the tuple space. Unlike Aardappel however there are no access

restrictions on hierarchical tuple spaces. Bonita also has first class tuple space values, but its focus instead is on providing more fine grained control on tuple space operations. It does this by splitting the operations up into less powerful and simpler ones that can work asynchronously (where the original Linda operations would block).

There are several declarative languages that have attempted to adopt Linda as their co-ordination model, Shared-ProLog [19] being a quite successful one. Shared-ProLog uses a tuple space as a basis for concurrency: it extends ProLog clauses by *guards* which are similar to tuple space in/out operations. In its semantics it is very similar to Linda, apart from the fact that it uses unification instead of pattern matching. Much like tree rewriting in Aardappel, the dynamically typed ProLog semantics are a great match for Linda, but it doesn't buy ProLog more than just concurrency and a cleaner global store than assert/retract (Shared-ProLog uses a single global tuple space).

Another ProLog variant, Bin-ProLog, only uses a Linda tuple space type construct as a simple replacement for assert-retract style global shared data, and doesn't implement multiple threads.

Astro-Gofer [24] tries to marry Gofer [41] with Linda. While yielding an interesting programming style for Gofer, Linda is not naturally at home with a language that relies on static typing and referential transparency, and the worst consequence of this is that state and concurrency semantics are hidden from semantics of the language, i.e. they are happening "magically" behind the scenes. ISETL-LINDA [25] is an earlier language from the same authors, which is more straight forward as (I)SETL[23] is an imperative language that supports dynamic typing.

Another related language is Gamma [38]. Gamma is related to Linda in the sense that it uses a multiset as a central structure in the language and that it tries to achieve parallelism through it. Gamma's approach is fairly different though: one creates programs by doing different kinds of function composition on (conditional) functions which can then apply to the (unstructured) data in the multiset. Concurrent entities in Gamma are the functions which rely on available data in the multiset to exploit parallelism. The biggest difference from Linda however, is the fact that Gamma is non-deterministic, i.e. it potentially has to try all combinations of data-items in the multiset with all functions to find a successful termination of the program. While this yields a potentially more expressive language, it also has terrible performance problems (it requires backtracking). For a more in-depth discussion, see [70].

8.3 Sharing

A well known attempt at marrying state with declarative programming is *monads* [42], implemented in the Haskell language [36]. Monads keep the referential transparency of the language intact, and as such don't offer true state but are a way to hide plumbing of state through the program code, allowing it to be easily modifiable and extendible. The price to pay for this is that all program fragments have to be concatenated using monad-friendly operators, which severely clutters code. Designing new monads and combining them with existing monad code is also sufficiently complex that it won't be a practical option for most programmers. As said, theoretically programs using monads are still referentially transparent, but complexity has gone up so much that it is hard to derive code clarity from it, which defies the objective.

There are several other systems that attempt to do functional state, [37] is a good starting point.

A technique to give a functional language some of the benefits of mutation without sharing is *linearity*: a system where every value has exactly one reference. If something has one reference to it, it can be safely mutated, as nobody else *sees* it. This allows for a whole range of things to be done more easily, more clearly, and often in particular: more efficiently.

The use of linearity in programming languages is discussed at length by Baker [6][7]. An example functional language with linear semantics is MARKOV [46]. A well known hybrid language is Clean [8]: Clean has a type system that includes linearness, and can do checking and inference on linearness. Linear values in Clean are used to efficiently implement stateful data structures, and in particular they allow clean and referentially transparent support for operating system features (for example, in Clean the concept of a file can be passed around as value because the system can prove that only one reference to it exists and allow it to be written to "in place"). A very similar language to Clean in the area of logic programming languages is Mercury[64] which enriches ProLog with a type system and a "mode system" which (like Clean) allows for checking and inference of properties other than types, such as linearity, free/bound variables, in/out arguments, deterministic predicates etc. Adding linearity to a type system for a (lazy) functional language is discussed in depth in [67]. Linear implementation of FFP-like systems was already discussed in [45].

A recent design for sharing in imperative languages is Balloon Types [3], which is a change to OO semantics to allow objects to shield themselves and all their child data

structures completely from the rest of the world, i.e. no pointers are allowed to enter the balloon but can only point at the root object of the balloon. This is a more solid design than merely relying on data hiding to shield child objects, as it is now enforced by the language. But it doesn't fundamentally change anything about the semantic model of the language, doesn't propose any new ways of organizing sharing in a different way, it merely restricts the status quo. Restriction isn't the solution, as for certain architectures / algorithms sharing is necessary, and will then still be messy.

8.4 Graphical languages

Of the whole family of graphical languages Aardappel falls in the category of graph rewriting languages. We leave aside icon rewriting languages (and picture/pixel rewriting languages such as Mondrian [44] and Visulan 3D [69]), as besides using the rewriting paradigm they don't have a lot in common with Aardappel: they use world structures not suitable for general purpose programming, and don't face the placeholder problem because they almost exclusively operate on constant values (abstraction features, when available, don't scale up). Two true graph rewriting languages are:

Chemtrains [9]. The data structure this language rewrites (the world) is a proper graph, with relations such as containment, proximity and connection. Rules are ordered to importance, and evaluation is rule driven (instead of data driven as is the case in most rewriting languages): the evaluator find those graph configurations in the world that match up with a certain rule (important rules first), and should there be more than one picks one at random to rewrite. Needless to say this is rather inefficient, but gives a programming model that is as intuitive as you can get (read some of the examples for some of the most readable code you will ever see). Sadly Chemtrains has the same drawbacks as the icon rewriting languages mentioned above in that all state of the world (all data in your program) is graphically represented on your screen at once, and as such limited to tiny simulations.

PROGRES [58] is a hybrid graphical / textual imperative language which uses graphical rewrite rules in combination which textual code and annotation. Unlike Chemtrains it rewrites full graph data structures of any size, and is therefore about the closest there is to a general purpose graphical rewriting language next to Aardappel. It's inefficiencies stem mainly from the fact that rewriting a graph is inherently more involved than rewriting trees.

Chapter 9

Conclusion

9.1 Conclusion

Mechanisms for shared access to data (in its various forms) are some of the most important issues in computer science: a lot of problems and research areas can be reformulated as trying to structure the sharing in a system to be cleaner, more optimal, less error prone, more secure or easier to maintain and organize. If it weren't for sharing, a lot of problems would be trivial, but also useless. Cleverly designed models of sharing can make all the difference to the effectiveness of a system, whether it's as local as a CPU or as big as the internet.

A system can be designed starting from many different perspectives, which then become the focal points of making a "good design". In too many cases sharing is something that is taken for granted and not used as the focal point of a design. This thesis has taken the opposite approach: design a new model of sharing and make the rest of the system be in tune with it.

In particular we focused on a particular form of sharing: sharing of mutable runtime values in programming languages. The resulting design, the tree space model of sharing, is not just an arbitrary set of restrictions or additions to one of the basic models we have seen so far in programing languages, instead it comes forth from the analysis of shared values and their observers (dependencies), and the Linda coordination language. Linda has always been stunning in its simplicity and effectiveness, and in hindsight this is not surprising any more once you see its 1:1 correspondence with the (shielded) set of observers in Aardappel. Aardappel's sharing model has such simple and clear

semantics that it is a basic model which has a natural place next to other basic models like imperative or functional.

Aardappel is a well rounded language design with the sharing model as its center. The features that make up the language can be said to be the minimal ones to make Aardappel into a comfortable and general purpose language, and fit well with the sharing model: Tree rewriting is one of the simplest models of sequential computation that is still very easy to program with.

To further stress the feasibility of the language design, a formal specification has been presented to show the simplicity of the semantics and ensure it is precise and unambiguous. On the practical side a full implementation of the language has been made, sparing no effort to make it a tool which can actually be useful in practise by giving it a professional editor / debugger / project manager integrated graphical development environment which uses modern technologies such as the Java virtual machine and RMI to give instant program execution, stand alone compiled programs and distributed computations. Throughout both language design and in the implementation an experimental graphical syntax was used, made possible due to the simple structure of the language. The purpose of the implementation was to validate the language design (the definition of the language in its present state was completed before implementation began, and took almost two years), and no changes to the design were needed as a consequence of practical experience with the language.

Future work would be almost exclusively in the realm of implementation. Some examples are: development of standard libraries, an optimizing compiler and type inferencer (as discussed in section 6.6), and a more high performance runtime system / Linda implementation.

Bibliography

[1] Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, Orlando, Florida, January 1991.

[2] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173, Portland, Oregon, January 1994.

[3] Paolo Sergio Almeida. Balloon types: Controlling sharing of state in data types. Technical report, Imperial College, 1997.

[4] Allen L. Ambler et al. 1997 visual programming language challenge. Technical report, IEEE, 1997.

[5] Stephen W. Bailey. Hielp user guide. Technical report, University of Chicago, 1993.

[6] Henry G. Baker. Lively linear lisp – 'look ma, no garbage!'. *ACM SIGPLAN Notices*, August 1992.

[7] Henry G. Baker. 'use-once' variables and linear objects – storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, September 1994.

[8] Erik Barendsen and Sjaak Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, June 1996.

[9] Brigham Bell. Chemtrains: A rule-based visual language for building graphical simulations. Technical report, US West Advanced Technologies, 1992.

[10] E. Berger. FP + OOP = haskell. Technical Report TR-92-30, University of Texas at Austin, Austin, TX, 1992.

[11] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML). *The World Wide Web Journal*, 2(4):29–66, 1997.

[12] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus linda. *Lecture Notes in Computer Science*, 924:66–??, 1995.

[13] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[14] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: a First Course*. The MIT Press, 1990.

[15] Nicholas Carriero and David Gelernter. Tuple analysis and partial evaluation strategies in the linda precompiler. In A. Nicolau D. Gelernter and D. Padua, editors, *Languages and Compilers for Parallel Computing*. Pitman, 1990.

[16] Nicholas Carriero and David Gelernter. Linda and message passing: What have we learned? Technical report, Yale University, Department of Computer Science, 1993.

[17] Nicholas Carriero, David Gelernter, and Jerry Leichter. Distributed data structures in Linda. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 236–242, St. Petersburg Beach, Florida, January 1986.

[18] Craig Chambers. The cecil language: Specification and rationale. Technical report, University of Washington, March 1993.

[19] Paolo Ciancarini. Blackboard programming in Shared ProLog. In A. Nicolau D. Gelernter and D. Padua, editors, *Languages and Compilers for Parallel Computing*. Pitman, 1990.

[20] J. Clark. Xsl transformations (xslt) version 1.0, w3c working draft. See http://www.w3.org/TR/WD-xslt, 1999.

[21] W. Clocksin and C. Mellish. Programming in prolog (3rd ed). Springer-Verlag, 1987.

[22] P.T Cox et al. Prograph: a step towards liberating programming from textual conditioning. In *Proc. 1989 IEEE Workshop on Visual Programming*, 1989.

[23] Robert B. K. Dewar. The SETL programming language. Technical Report, 1979.

[24] A. Douglas, N. Roejemo, C. Runciman, and A. Wood. Astro-Gofer: Parallel functional programming with co-ordinating processes. *Lecture Notes in Computer Science*, 1123:686–??, 1996.

[25] Andrew Douglas, Alan Wood, and Antony Rowstron. ISETL-LINDA: Parallel programming with bags. Technical report, University of York, 1995.

[26] Brian Foote and Joseph W. Yoder. Big ball of mud. In *Fourth Conference on Patterns Languages of Programs*, 1997.

[27] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):28–39, September 1991.

[28] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, February 1992.

[29] A. Goldberg and D. Robsen. *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, 1983.

[30] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, 1996.

[31] Albert Graef. The q programming language. In *Musikinformatik und Medientechnik*. Johannes Gutenberg-Universitaet Mainz, 1992.

[32] Peter Grogono and Patrice Chalin. Copying, sharing, and aliasing. In Colloquium on Object-Orientation in Databases and Software Engineering, Montreal, Quebec, 1994.

[33] W. Hasselbring. Combining SETL/E with Linda. Technical report, Computer Science/Software Engineering Department, University of Essen, Germany, June 1991.

[34] P.R.H. Hendriks. ASF system user's guide. Technical report, Centrum voor Wiskunde en Informatica, 1988.

[35] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.

[36] Paul Hudak, Simon Peyton Jones, and Philip Wadler et al. Report on the programming language haskell, A non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[37] Paul Hudak and Dan Rabin. State in functional languages an annotated bibliography version 0.0. Technical report, Yale University, June 1993.

[38] D.Le Métayer J.-P. Banâtre. Gamma and the chemical reaction model: ten years after. In *Coordination programming: mechanisms, models and semantics, IC Press*. World Scientific Publishing, 1996.

[39] R. Jellinghaus. Eiffel linda: An object-oriented linda dialect. *SIGPLAN Notices*, 25(12):70–84, December 1990.

[40] M. Jones and J. Peterson. The nottingham and yale haskell user's system. Department of Computer Science, University of Nottingham, 1997.

[41] Mark P. Jones. The implementation of the Gofer functional programming system. Technical report, Yale University, 1994.

[42] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84, Charleston, South Carolina, January 1993.

[43] J. Kodosky et al. Visual programming using structured data flow. In *Proceedings of the IEEE Workshop on Visual Languages*, 1991.

[44] Henry Lieberman. Mondrian: A teachable graphical editor. In Stacey Ashlund, Ken Mullet, Austin Henderson, Erik Hollnagel, and Ted White, editors, *Proceedings of the Conference on Human Factors in computing systems*, pages 144–144, New York, April 24–29 1993. ACM Press.

[45] Gyula Magó. Copying operands versus copying results: a solution to the problem of large operands in FFP's. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, 1981.

[46] Nikolai Mansurov. MARKOV: First-order language without garbage collection. In *Implementation of Functional Languages*, 1994.

[47] D. C. J. Matthews. Poly manual. *ACM SIGPLAN Notices*, 20(9):52–76, September 1985.

[48] J. McCarthy et al. LISP 1.5 programmer's manual. Technical report, The MIT Press, 1965.

[49] James E. Narem Jr. An informal operational semantics of C-Linda V2.3.5. Technical report, Yale University, 1989.

[50] J. V. Nickerson. Visual programming: Limits of graphic representation. In *IEEE International Symposium on Visual Languages*, 1994.

[51] Michael J. O'Donnell. *Equational logic as a programming language*. MIT Press series in the foundations of computing, 1985.

[52] Stephen M. Omohundro. The Sather language. Technical report, International Computer Science Institute, 1991.

[53] Wouter van Oortmerssen. The aardappel programming language: A gentle introduction. In *Proceedings of the 1997 IPISA Conference*, 1997.

[54] G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers*, volume 46: The Engineering of Large Systems. Academic Press, 1998.

[55] John Peterson and Mark Jones. Implementing type classes. *SIGPLAN Notices*, 28(6):227–236, June 1993.

[56] Alex Repenning. Bending icons: Syntactic and semantic transformations of icons. In *IEEE International Symposium on Visual Languages*, 1994.

[57] A. Rowstron and A. Wood. Bonita: a set of tuple space primitives for distributed coordination. In *Proc. HICSS30, Sw Track*, pages 379–388, Hawaii, 1997. IEEE Computer Society Press.

[58] Andy Schrr. Progres, a visual language and environment for programming with graph rewrite systems. Technical report, RWTH Aachen, 1994.

[59] Scientific Computing Associates, Inc., New Haven, CT, USA. *FORTRAN-Linda Reference Manual*, 1993.

[60] David Sherman and Robert Strandh. An abstract machine for efficient implementation of term rewriting. Technical report, University of Chicago, 1990.

[61] David Sherman and Robert Strandh. Partial evaluation of intermediate code from equational programs. Technical report, University of Chicago, 1990.

[62] David J. Sherman. Sharing common subexpressions in EM code programs. Technical report, University of Chicago, 1990.

[63] D. C. Smith, A. Cypher, and J. Spohrer. KidSim: Programming agents without a programming language. *Communications of the ACM*, July 1994.

[64] Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In Proceedings of the Australian Computer Science Conference, pages 499–512, February 1995.

[65] Geoff Sutcliff and James Pinakis. Prolog-Linda: An embedding of Linda in muProlog. Technical report, Department of Computer Science, The University of Western Australia, Nedlands, 6009, Western Australia.

[66] Ulf Dahlen, Department of Computer and Information Science, Linkoping University, Linkoping, Sweden. *Scheme–Linda*, September 1990. EPCC-TN90-06.

[67] D. Wakeling. Linearity and lazyness. Technical report, University of York, 1991.

[68] D. Wakeling. Vsd: a haskell to java virtual machine code compiler. Intl. Workshop on Implementing Functional Languages (IFL '97), St Andrews, Scotland, Springer-Verlag LNCS 1467, 1997, pp. 39–52., 1997.

[69] Kakuya Yamamoto. Visulan: A visual programming language for self-changing bitmap. Technical report, Kyoto University, 1995.

[70] G. Zavattaro. On the incomparability of gamma and linda. Technical Report SEN-R9827, CWI - Centrum voor Wiskunde en Informatica, 1998.

Appendix A

The AardED readme file

Aardappel Programming Environment
User Manual

Wouter van Oortmerssen

Installation
============
The Aardappel Editor is written in Java, and the compiler generates Java applications so you will need a Java SDK or JRE installed on your system (a Java enabled browser won't do). Get it from:

http://java.sun.com/jdk/

To install AardEd itself simply unzip it to any place you like (with directories). If you have Java correctly installed you should be able to run AardEd using the included aarded.bat (windows) or aarded.sh (any unix) scripts. You may need to modify these to use "javaw" or "jre" instead of "java", depending on what kind of Java you have installed.

Using AardEd

==========

Note well: this text is there to provide all practical information not included in my thesis, as such it does not explain the language. To be able to use AardEd sensibly you are advised to read atleast chapters 3 and 6 of my thesis, available from where you got this, or:

http://www.ecs.soton.ac.uk/~wvo96r/aardappel/

A few sample Aardappel projects have been included with AardEd that you can load, browse through, and execute, to get a feel for the system. Use "open" from the project menu and go to the "projects" folder in the top directory. Load "project.ap" from any of its subdirectories. "factorial" or "qsort" are good starting points.

To create your own project, simply create a new directory below the projects dir (or anywhere else) and use "new" from the project menu to save a "project.ap" file there. Then add new bags & modules to your project and edit them. To save, use "save all" from the menu, which saves all bags & modules and your project file. Quitting and restarting AardEd will automatically load the last project you were working on.

Editing programs

==========

Again, most should become appearent from what is in the thesis and some experimenting, but here are some general hints & tips to get started.

The best way to create any tree is drag and drop: start at the root and drag/replace children as you go. If you drag trees from another rule they will be copied, if you drag them from pattern to expression part you will create a placeholder. If you want to copy something from another window, use copy & paste.

An alternative method to create trees is typing the name of the root atom in the string widget at the top of the window. If it's

an existing tree with rules for it, Aarded will fill in the children for you. To add children to a tree, use the the right mouse button menu (ctrl-A).

Creating your own icons

You can associate a graphical representation with any atom you use in your programs. Simply create the picture in a paint program, save it in gif format using transparent background if applicable, and put it the "atomimages" subdirectory under <name>.gif, where <name> is equal to the atom you want replaced (for operator symbols use their ascii value). AardEd will automatically scan these pictures when it starts up and render your atoms accordingly.

Similarly if certain icons confuse you, all that you need to do is remove them from that directory (currently the only icons I made are some operators, "lambda", and a couple for the qsort example).

Running an Aardappel program separate from AardEd

To give compiled Aardappel programs to other people without the environment, simply create a zip or jar that includes the following files:

CompiledAardappel.class
AardappelRuntime*.class
r/*.class

and maybe a script file that runs

java CompiledAardappel

CompiledAardappel.class is simply the last compiled expression, so make sure you select "compile standalone" on the appropriate top level expression / bag before you copy these files.

Distributing programs over a network
==========

To set up a machine as a server that can help out in Aardappel computations, it needs to have a copy of these files:

AardappelRuntime*.class
r/*.class

AardEd uses RMI to communicate with the servers, so you need to have "rmiregistry" running. This should be part of any Java installation.

You can then launch the server as follows:

java r/AardappelServer localhost

If everything went ok, this will then tell you that it is ready to participate in computations. On your client machine you can now run your Aardappel program as standalone and specify the ip addresses / names of any servers on the command line:

java CompiledAardappel server1 server2 ...

if everything went ok your program should now run distributed over all available servers and the client!

You can also start distributed computations directly from AardEd, but depending on the way your system is set up there may be problems with that. The infrastructure for it is in place, look at the source for Compiler.compile() if you are interested.

In most cases the above is too simple to be true, and trying to run a distributed computation out of the box will result in various obscure security exceptions being thrown. RMI is subject to the RMISecurityManager, which under Java 1.2 or greater gets its permissions from the java policy files (they don't exist on 1.1, so it

should be much easier to run the distributed runtime system on that), which in most default installations don't allow enough for the runtime system to work.

To fix this requires you fiddle with the policy files in your Java installation. You can take the lazy route and put something like (replace <path> with where you installed AardEd):

```
grant codeBase "file://<path>/aarded/-" {
permission java.security.AllPermission;
};
```

at the top of java.policy (depending on your system, the one in jre/lib/security, or you can create one in your home directory or in the aarded directory, in which case you may need to point to it from the java.security file). Alternatively you can read up on Java security and try to grant permissions more precisely :)

Not implemented functions

from the menu: "deploy" (use compile standalone in the treeview editors instead).

Source code

This release comes with full sourcecode, which you can browse out of mere interest or enhance yourself.

One of the first thing you may want to add if you intend to write some Aardappel programs are more builtin functions: the basic ones are there but there are bound to be ones missing. To do this, add a function to the existing ones in AardappelRuntime.java, like:

```
public static Rval builtin_rule_myfunction_2(Rval a, Rval b) { ... };
```

(the 2 is the number of arguments). Then in the constructor in Compiler.java add a line like:

```
builtin("myfunction",2);
```

Now you can use trees with root atom "myfunction" and two children and have this function evaluated instead. Look at other builtin functions to see how to access an Rval. Caveat: if you add functions that have side effects you will have to use AardappelServer.marshall() to make sure they are executed on the client, if you want them to behave correctly in a distributed program (look at "plot" for how to do this).

General guide to the source code files:

the runtime system consists of AardappelRuntime.java + all the files in the "r" package, most of which implement various runtime representations of Aardappel values. Most interesting is the bag, which implements Linda (see RbagTS.java)

The project editor is in Mainwin.java and Project.java, and the graphical editor is in Treeview.java, all the subclasses of Code.java (Atom/Int/Bag/Tree/Rules), and the various *info.java classes.

The compiler is in Compiler.java, Codegen.java (a generic interface to the jas package) and Sym.java (contains the functionality for compiling all rules with a certain root atom).

most code should prove to be easily modifyable and maintainable.

Future versions, legality status etc.
==========

I would like to improve AardEd, especially now it finally got somewhere: all raw functionality is in, but it could do with a lot of improvements. I can't make any promises however, since I have taken on a quite demanding job, so don't come bug me for new releases or fixes

of certain problems.

Program & source are released under the GNU public license (read http://www.gnu.org/copyleft/gpl.txt). If you make an improved version of AardEd I would love to hear about it.

Wouter van Oortmerssen

wvo96r@ecs.soton.ac.uk
wouter@amiga.com
wvo@acm.org