Program Metamorphosis

by

Christoph Reichenbach

M.Sc., University of Colorado at Boulder, 2004

A dissertation submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2009

This dissertation entitled:
Program Metamorphosis
written by Christoph Reichenbach
has been approved for the Department of Computer Science

Dr. Amer Diwan

Dr. Jeremy Siek

Dr. Ken Anderson

Dr. James Martin

Dr. Bor-Yuh Evan Chang

Date

The final copy of this dissertation has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Reichenbach, Christoph (Ph.D., Computer Science)

Program Metamorphosis

Dissertation directed by Prof. Dr. Amer Diwan

Today's computer programs are logical constructs described in human-readable form as source code. Manipulating such source code can be challenging even to trained programmers, as modifications in one location can affect program behaviour described in other locations in nontrivial ways.

While there are many programs that help programmers modify source code to evolve programs, these tools are typically very limited in their scope: they either give few (if any) guarantees to the programmer about what program behaviour they will affect, or they promise near-total behaviour preservation but accomplish this by means that the programmer may not always agree with.

In this work we introduce a novel mechanism for program evolution. Our principal idea is that we capture program behaviour in the form of program models and then permit the programmer to apply an arbitrary number of changes to the program. After these changes, we compute behavioural differences between the original program behaviour and the modified program behaviour. Programmers can then choose which changes they accept and which changes they wish to undo, and how.

We find that our approach subsumes and extends on the existing approaches for transforming programs with near-total behaviour preservation, with stronger guarantees of behaviour-preservation even for a language for which such guarantees are traditionally hard to obtain (Java). Furthermore, we show that for another language (SML) our behaviour-preservation guarantees can be near-absolute whilst permitting complex multi-step transformations that speed up program execution (algorithmic optimisation).

Dedication

This dissertation is dedicated to all craters named after computer scientists on the side of the moon that opposes the earth. Craters on the fringe are included iff at least 20% of their interior surface is visible from one point of the earth.

Acknowledgements

The author would like to thank Devin Coughlin, for providing the implementation of the Java prototype, as well as Walter van Niftrik, Lena Andersson, Mikael Wilander, Matt Hargett, Philipp Wetzler, Daniel von Dincklage, Christian Doerr, Gary Yee, Simon Thompson, and the anonymous POPL, PLDI, ECOOP referees for their valuable feedback to numerous aspects of the ideas and the writing presented herein.

Contents

Tables

Figures

Chapter 1

Motivation

Software evolves. Changing requirements, deeper understanding, and new libraries and tools all urge programmers to re-visit their earlier design decisions. Present-day software development methodologies encourage programmers to treat such reappraisal as fundamental, i.e., to "embrace change" [Bec00, FBB^{+}99]. However, programmers still have very limited interactive tool support for this task:

- *Manual transformation* — programmers can use a text editor to modify a program,
- *Automatic program transformation* — programmers can generate a new program from a specification or from an existing program,
- *Program refinement* or *synthesis* — programmers can use a specialised methodology to iteratively derive a program from a formal specification, and
- *Refactoring* — programmers can apply specialised program transformations that are equipped with a guarantee of behaviour preservation.

These techniques vary widely with respect to both the *transformative power* and the amount of *behavioural guarantees* they provide.

		Flexibility in	
Approach	**Behaviour**	**Refactoring**	**Evolution**
Manual editing	—	any	any
Automatic Transformation	—	any	any
Program Refinement	external spec	with proof	none
Traditional Refactoring	equal	one-step	none

Table 1.1: Comparison of program modification techniques.

Table 1.1 summarises how these four techniques compare, first on the behavioural guarantees they offer, and then on their flexibility when updating programs.

In the table above, column **Behaviour** shows that only program refinement and refactoring offer any guarantees about the *behaviour* of the program after transformation[1] : program

[1] While all approaches offer some amount of *safety* guarantees from the programming languages' type checking and additional program checking tools (such as lint), these guarantees only affect the well-formedness of the program.

refinement guarantees that the program still satisfies the same high-level specification as before (‘external spec,’ in Table 1.1), while refactoring guarantees that the program after and the program before transformation have exactly the same behaviour (‘equal’).

The two **Flexibility** columns contrast the approaches on the kinds of transformations they support: behaviour-preserving structural transformations (**Refactoring**) and behaviour-extending or -altering program transformations (**Evolution**). Only manual editing and program transformation offer full flexibility for both tasks, but this is natural, as any mechanism that provides safety guarantees must place restrictions on possible program modifications. For program refinement, the restrictions depend on the refinement calculus the system is based on; in principle, any modifications may be permissible as long as they ensure that the program satisfies the high-level specification. For traditional refactoring tools, the restrictions are more severe: the refactoring must, in a single step, preserve all program behaviour.

We note that these four mechanisms are insufficient for at least three practical scenarios:

- *Multi-step refactoring.*
  Consider the following problem:

  Traditional refactorings must atomically preserve behaviour within one transformation. For example, an ‘inline definition’ refactoring must not cause name capture. If a user requests an ‘inline definition’ refactoring that would cause name capture, the refactoring system must either deny this request or implicitly perform other refactorings, e.g. to rename variables to avoid name capture.

  In the first case, the refactoring system ignores a user request, which is inconvenient, and in the second case, the refactoring system changes more than the user requested, which can be confusing. A straightforward solution would be to split the refactoring into multiple steps that together preserve behaviour, but existing tools offer no support for this.

- *Custom datatype replacement.*
  Consider the following problem:

  Programmers sometimes replace datatypes to speed up their programs. For example, they might replace an array by a custom integer map implementation that is faster or uses less memory in a particular scenario. However, in general no tool can know that the two datatypes are equivalent — and in which way they are equivalent — without user-specified information.

- *Partial behaviour preservation.*
  Continuing on the above problem, assume that the integer map returns a default value (such as `null` in Java) when a user accesses an invalid index, rather than raising an exception (as a Java array might).

  Then, replacing the array by the integer map is a potential behavioural change, since we might no longer raise an exception that we used to raise but might now (for example) raise a different exception. However, in many programs an exception during element lookup either cannot occur or signals a fatal error condition — meaning that programmers have no reason to eschew the possible speedup for a ‘minor’ change in exception behaviour.

> Other examples of partial behaviour preservation abound: instrumentation, changing public APIs, re-ordering I/O operations, and many other forms of algorithmic optimisation all are tasks that benefit from partial behaviour preservation.

In this work, we show that we can overcome these limitations with a small twist to the existing idea of refactoring, without weakening our behaviour-preservation guarantees:

Thesis Statement

Interactive multi-step program transformation, when augmented with suitable mechanisms for determining change in program behaviour, can offer behaviour-preservation guarantees that are at least as strong as those of existing single-step program transformation techniques (specifically refactoring) while offering greater flexibility and a wider range of application.

Chapter 2

Introduction

Software evolves in many ways. To build a useful tool for software evolution, we should therefore attempt to support as many kinds of behavioural evolution as possible. However, we are not aware of a complete catalogue of forms of source code change that programs undergo during software development. The closest results would appear to be Dig and Johnson's analysis on API changes [DJ05] and, in particular, Rupp-Greene's survey [RG09] on this matter. While both point to refactorings as a significant contributor to program change, Rupp-Greene's results suggest that at least during some stages of program development, refactorings play a less significant role than behaviour-evolving steps.

Clearly, tool support for the task of adding new modules and new functionality will be limited — unless we mimic program refinement and allow separate specifications, we have no means for checking whether a new module does what it is expected to do. We can however ensure that new code 'plays nicely' with the old code, e.g. by ensuring that side effects during any new code's initialisation phase do not interfere with side effects from existing code, or that new definitions do not mask existing definitions. We can further help in integrating new code into the program. For example, if the user builds a new datatype that is meant to supplant an existing datatype, we can offer facilities to replace one by the other.

On top of behaviour-extending changes, we should also support all the behaviour-preserving changes that existing approaches (specifically, automatic refactoring) have to offer. In principle, existing approaches to automatic refactoring (i.e., one-step behaviour-preserving program transformations) can offer a wide range of program transformations. In practice, these transformations are more limited; clearly it makes more sense for us to draw inspiration from what refactoring tools *could* provide than from the more limited set of what they *do* provide.

Thus, we review some of the shortcomings of present-day automatic refactoring, beginning with the limitations of total behaviour preservation (Section 2.1). We then debate the actual validity of the 'behaviour preservation' guarantees given by existing refactoring systems in Section 2.2. Afterwards, Section 2.3 discusses the restrictions imposed by atomicity of behaviour-preserving transformations in existing systems.

2.1 Limitations of total behaviour preservation

Consider one example of a transformation: when we profile program performance, we often find it helpful to understand how often a specific function is called, and how calls are distributed among call sites. Often, we have to implement this instrumentation by hand.

Since such transformations introduce and modify global program state and typically include some kind of output or, at least, API change (to make the gathered information accessible), they are not fully behaviour-preserving.

There are many other examples of *partially* behaviour-preserving changes:

- Other forms of information-gathering instrumentation, such as
  - Execution time gathering
  - Branch coverage statistics gathering
  - Logging and debug messages

  For such instrumentation, we must keep some kind of record of the gathered information. Behavioural changes must be restricted to the addition of this record and to manipulations of this record.
- Externally visible API changes. No behavioural or API changes are allowed beyond the changes explicitly mandated by the user.
- Elimination or introduction of possible values of an enumeration (or algebraic datatype), and of handlers in for the introduced/eliminated value. Other handlers and all unrelated behaviour must be preserved.
- Adding a new option to a type less structured than enumerations, such as a list of strings denoting options (e.g. representing command-line or configuration parameters). Existing behaviour must be preserved as the "default" behaviour.
- Deriving any kind of additional functionality, such as visualisation or hashing functions for algebraic datatypes. All other functionality must remain untouched.
- Performing changes which sacrifice accuracy for performance, or increase accuracy (such as changing the type of a single-precision floating point variable to a double-precision floating point value).
- Changes that affect only part of the program semantics, such as printing a value whenever a function is called while leaving the function's result unchanged.

All of the above transformations are ruled out by the full behaviour preservation guarantee given by refactorings, even though they are practical and useful examples of program transformations. Moreover, all of the above can be sensibly 'constrained' in that we can find nontrivial guarantees of behaviour preservation to accompany them.

2.2 Limitations in behaviour preservation guarantees

In 'Refactoring: Improving the Design of Existing Code' [FBB+99], Fowler requires refactorings to "preserve external behaviour". This requirement leaves room for interpretation, we argue that 'external behaviour' should at least include user-visible output.

A quick survey reveals that even this basic requirement is not guaranteed by refactoring implementations in state-of-the-art refactoring systems [SDF+03, Sof].

Consider the '*Change Method Signature*' refactoring, which can be used e.g. to swap two parameters in a method. Correspondingly, the same parameters will be swapped automatically at all call sites. This ensures that the resulting values are passed into the program in the correct order, but consider the Java program in Figure 2.1: This program will print the result of summing up the return values of a call to `foo.getX()` and `foo.incX()`. The first call will

```
public class Foo {
  private int x = 0;
  public int getX() { return x; }
  public int incX() { return ++x; }
};
public class Bar {
  public static int encode (int a, int b) {return a + b;}
  public static final void main(String [] _) {
    Foo foo = new Foo();
    System.out.println(encode(foo.getX(), foo.incX()));
  }
};
```

Figure 2.1: A Java program that will behave differently after swapping parameters.

return the x field of the foo it is applied to, while the latter will first increase this field and then return it. Since Java guarantees the order of evaluation of method parameters from left to right [GJSB00], this means that we will first call getX() and then incX(), then print the sum of the results, which is 1. However, if we swap these parameters, incX() will be evaluated first, meaning that the getX() call will also return 1, printing a total result of 2. Therefore, such an application of '*Change Method Signature*' should be disallowed. However, the implementation of this refactoring in Eclipse 3.4.1 such uses of side effects and therefore permits this transformation, changing external program behaviour.

Such problems are not isolated incidents; below is a number of other applications of refactoring implementations we have observed which fail to preserve program behaviour:

- IntelliJ IDEA 5.1's '*Change Signature*' refactoring suffers from the same problem as shown above for Eclipse.
- The '*Introduce Parameter*' refactoring in Eclipse 3.4.1 will introduce new parameters to methods and constructors, and amend all call sites to pass in the original parameters. Since Eclipse uses textual substitution for this part of the transformation, the resulting calls may invalidate the program or, worse, subtly change behaviour through name capture.
- Most, if not all, existing refactorings for Java subtly change program behaviour, since the observable behaviour of a Java program includes all information that can be gathered about existing classes via reflection. Therefore, any renaming or type change of a public field, method, or constructor changes the observable behaviour.

Software developers should thus remain wary of any refactorings performed by existing and even well-tested tools: Jim Wagner at internetnews.com[1] claims that the Eclipse development environment had been downloaded about 18 million times by early 2004, indicating a significant potential for user feedback to be collected and incorporated by 2009. (According to the official Eclipse website, more recent releases have been downloaded more than 3.5 million

[1] http://www.internetnews.com/dev-news/article.php/3314361

times[2] , so this number may be a plausible estimate of sum of all downloads of all releases up to 2004.)

For this reason, Fowler (and Kerievsky) complement the refactoring process through testing [FBB+99, Ker04]. This precaution alleviates, but does not solve the problem: Testing can only exercise a finite number of test cases, whereas many programs accept a theoretically infinite number of possible inputs.

2.3 Limitations of one-step behaviour preservation

Consider the following SML program fragment:

```
val count = 23
val w = compute_count(42)
val x = count + 2
val y = x + w
```

Assume that a programmer wishes to apply a *rename* refactoring [FBB+99], to rename w to count. This transformation is problematic because it results in a name capture and the expression count + 2 ends up using the wrong count:

```
val count = 23
val count = compute_count(42)
val x = count + 2
val y = x + count
```

Refactoring systems (such as HaRe [LRT03,LTR05], Eclipse [SDF+03], or IntelliJ [Sof]) deal with this problem in one of two ways: They either

- Disallow the refactoring, or
- Apply a built-in heuristic to perform additional transformations (such as implicitly renaming the previously existing count to count1).

Neither approach is ideal. Disallowing a user-requested transformation is inconvenient, and applying implicit refactorings means that the program is changed in ways which the user had not intended.

2.4 Challenges

This chapter highlighted and implied a considerable set of missing functionality in existing program evolution tools. Our approach permits much, perhaps all of this functionality. To illustrate this, we consider five challenges that summarise this missing functionality:

Challenge #1: *Multi-step transformations.* We show that our approach allows multi-step transformations, permitting renamings such as the one we sketched above (Chapter 3) and further facilitating refactorings that elude all existing refactoring tools (Section 5.1).

Challenge #2: *Safe behaviour evolution.* Behaviour evolution requires us to detect behavioural change, but we also wish to allow such change. In Chapter 3 we show that detecting change and allowing users to accept this change is a fundamental part of our system.

Challenge #3: *Substitute custom datatypes and algorithms.* User-defined datatypes and algorithms are often meant to supplant library datatypes and functionality. In Section 5.2

[2] http://www.eclipse.org/downloads/packages/release/ganymede/sr2

we show how we can support algorithmic optimisation based on user-specified datatype or algorithm equivalences.

Challenge #4: *Handle side effects separately from value results.* When we integrate new functionality (e.g., instrumentation), we may wish to distinguish changes to side effects from changes to value results. In Section 6.2.3, we show how our approach can support this idea.

Challenge #5: *Absolute behaviour preservation.* In Appendix D we show how one of our program models guarantees that we detect all behavioural changes (except for changes in termination behaviour) while suppressing many false negatives in an effectful and representative subset of Standard ML.

Chapter 3

Overview

To understand our approach, *program metamorphosis*, it is perhaps best to contrast it to its closest relative among the existing approaches to machine-supported program transformation, *automatic refactoring*. For contrast we will use the term *traditional refactoring* to the established practice of one-step behaviour-preserving transformations.

3.1 From refactorings to program metamorphosis

In the following, we recapitulate the traditional implementation strategy for refactoring, illustrate the shortcomings of this strategy in more detail, and contrast it with program metamorphosis.

3.1.1 How refactorings work

Abstractly, a refactoring is a pair $\langle P, t \rangle$. P is a safety precondition that determines whether or not the refactoring is applicable to a given program. t transforms the program (e.g., renames a variable).

Since refactorings must preserve behaviour, P must ensure that the program has the same behaviour before and after applying t:

$$P(p) \implies [\![t(p)]\!] = [\![p]\!]$$

where $[\![-]\!]$ assigns a program its behaviour. Refactoring implementors typically implement P by considering the possible ways in which t may alter program behaviour and then design P to detect these. P's implementation then relies on one or more program analyses that uncover relevant properties about the program. Thus, internally P is the following:

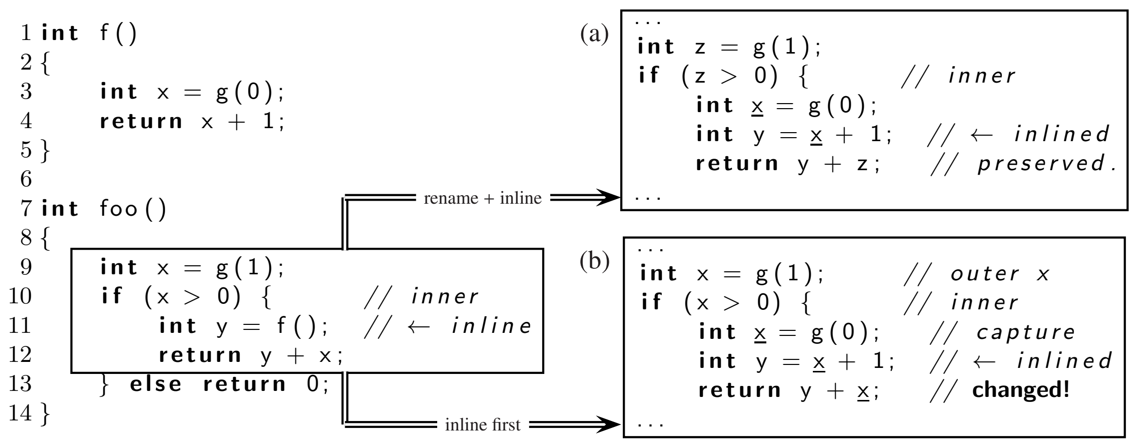
$$P(p) \implies c_P(\mathsf{properties}(p))$$

where $\mathsf{properties}$ computes all properties that are relevant to the refactoring and some check c_P determines whether the properties will guarantee behaviour preservation.

Since $\mathsf{properties}$ is a collection of program analyses, and fully precise program analyses are undecidable in general, refactoring designers are forced to make a decision: if they construct a conservative implementation of $\mathsf{properties}$, they will have a sound refactoring system but may disallow a number of correct refactorings. On the other hand, if they construct an unsound $\mathsf{properties}$, they may allow a large number of transformations, but will not be able to guarantee behaviour preservation. While existing refactoring tools (with the possible exception of HaRe [LTR05]) are not conservative, conservativeness (and therefore guaranteed behaviour preservation) remains the underlying motivation behind refactoring, and therefore an ideal for all refactoring tools.

3.1.2 Multi-step transformations and refactorings

However, this ideal is sometimes inconvenient. This inconvenience manifests, for example, with refactorings that do not commute. Consider method inlining in C in Figure 3.1:



```
1 int f()
2 {
3     int x = g(0);
4     return x + 1;
5 }
6
7 int foo()
8 {
9     int x = g(1);
10    if (x > 0) {          // inner
11        int y = f();      // ← inline
12        return y + x;
13    } else return 0;
14 }
```

(a)

```
...
int z = g(1);
if (z > 0) {          // inner
    int x = g(0);
    int y = x + 1;    // ← inlined
    return y + z;     // preserved.
...
```

(b)

```
...
int x = g(1);         // outer x
if (x > 0) {          // inner
    int x = g(0);     // capture
    int y = x + 1;    // ← inlined
    return y + x;     // changed!
...
```

Figure 3.1: Function inlining in C.

Assume that we are trying to inline function f into line 11: this would copy variable x from line 3 into function foo. foo already has a variable x, so we might want to rename one of them to z first. If we rename and then inline, we get output (a), which will compute the same result as before (here, we inlined f's x for illustration purposes). But if we inline first, we get output (b), which produces a different program result! If we now try to rename one of the x, then this will no longer help us: we have already captured the outer x.

In practice, it is hard for users to predict all the names captured by an inlining transformation: it would be easier to inline first and sort out problems later, when we can directly observe any conflicts.

More severely, some refactorings are inapplicable to the problems they are meant to solve. Consider moving two Java methods from class A to class B, as in Figure 3.2.

With a '*Move Method*' refactoring, we expect this transformation to require two steps, one for each individual move. But if we now try to move isOdd from A to B, isOdd and isEven will no longer be in scope of each other — the program will be ill-formed, meaning that we have not preserved behaviour (hence, we cannot permit this refactoring). Consequently, we must move both simultaneously to preserve behaviour.

```
class A {
    static bool isOdd(int i) {
        if (i = 0)
            return false;
        else return isEven(i - 1);
    }

    static bool isEven(int i) {
        if (i = 0)
            return true;
        else return isOdd(i - 1);
    }
}
class B {
    ...
}
```

Figure 3.2: Moving mutually recursive methods in Java.

Existing refactoring systems either ignore these problems, completely disallow such transformations, or try to address them by using ad-hoc solutions: For example, Eclipse's 'Inline Method' refactoring will implicitly rename 'offending' variables if they would otherwise cause name capture, using sound but arbitrary heuristics to pick variables and new names.

As an alternative to these approaches, we could explicitly support multi-step transformations. Composing traditional refactorings [KK04] will not allow us to resolve circular dependencies as in the above, but composing transformations *separately* from predicates would. For example, we might compose refactorings $\langle P_1, t_1\rangle, \ldots, \langle P_n, t_n\rangle$ as

$$\langle \mathsf{composeP}(\langle P_1, t_1\rangle, \cdots, \langle P_n, t_n\rangle), t_n \circ \cdots \circ t_1\rangle$$

where composeP constructs some new predicate that covers the full sequence of transformations.

However, this approach is still undesirable: it requires users to think through the full sequence of refactorings in advance, since each refactoring's preconditions may require additional preceding refactorings.

3.1.3 Towards program metamorphosis

Our approach solves the above problem by abandoning the traditional approach to refactoring. Program metamorphosis is based on two key insights:

(1) Using *postconditions instead of preconditions* allows users to experiment, rather than having to predict the outcome of a sequence of transformations: if we check for behaviour preservation *after* transforming, we can visualise the updated (possibly incorrect) program and allow users to pick the next transformation with complete knowledge of the effect of the preceding transformations.

(2) We can construct an appropriate postcondition by *logical decomposition of preconditions*: this allows us to build a "composeP" function for postconditions.

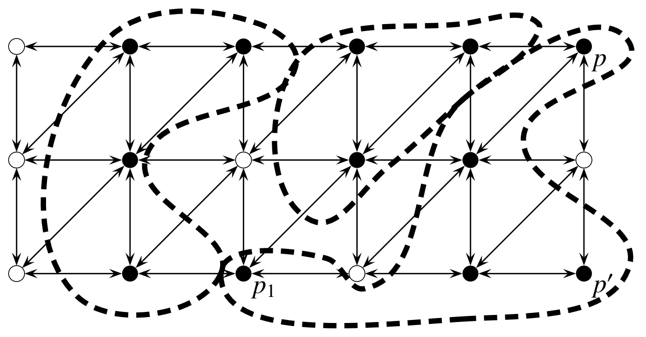


Figure 3.3: Abstract view on program metamorphosis.

The following equation summarises our first insight, for a postcondition Q

$$Q(t(p)) \implies [\![t(p)]\!] = [\![p]\!]$$

As befits a postcondition, we apply it to the transformed program, rather than to the original program.

To see the second insight, recall the *intuitive* idea behind preconditions and postconditions: these conditions ensure that the program has the same behaviour before and after the transformation, and is well-formed before and afterwards. In short,

$$P(p) \iff Q(t(p)) \iff V(p) \land (p \equiv t(p)) \land V(t(p))$$

where V ensures well-formedness and $(\equiv)$ approximates behavioural equivalence between two programs.

Preconditions are not normally implemented as instances of the above formula, though we show in our later experiments that this is not a limiting factor in practice.

If we take this idea one step further, we arrive at program metamorphosis. First, observe that the meaning of program validity is generally independent of which refactoring it is implemented for. Secondly, we can meaningfully combine the equivalence relations we find within disparate refactoring preconditions. We can now decouple equivalence and validity checking from applying program transformations: this is program metamorphosis.

To visualise the benefits of this approach, consider Figure 3.3: each solid edge here represents a program transformation. Programs are equivalent iff they are elements of the same equivalence class (marked by dashed lines). In this picture, refactoring is precisely the process of moving along the solid edges in this graph from one point in an equivalence class to another in the same class ("with the same behaviour"). In program metamorphosis we may choose any sequence of transformations $\bar{t} = t_1 \circ \cdots \circ t_n$ such that

$$Q(\bar{t}(p)) \iff V(p) \land (p \equiv \bar{t}(p)) \land V(\bar{t}(p))$$

In particular, we can traverse through other equivalence classes (programs with different behaviour) and even through ill-formed programs. By contrast, with classical refactoring we must

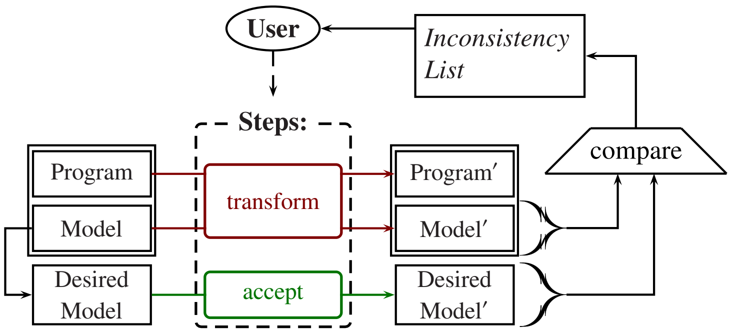


Figure 3.4: Program metamorphosis process.

choose $\bar{t}$ to be a single transformation. If we want to refactor p to p' in Figure 3.3, we thus need one extra step to reach p', and we can never reach p_1, even though it is semantically equivalent to p.

3.2 Implementing program metamorphosis

The well-formedness predicate V is no harder to implement than a frontend for whichever object language we choose, but the equivalence relation ($\equiv$) presents a greater challenge, as do our transformations: how can we 'correctly' transform an ill-formed program, and what does that mean? And how can we compare the behaviour of the current program with a long-gone program?

Clearly the program by itself cannot be sufficient to represent all behaviour. We thus enrich the object program (i.e., the program we are transforming) with a *program model* that we maintain in parallel to the program itself. This program model captures additional semantic information about the object program. We split it into two halves: the *current program model*, capturing the current object program's behaviour, and the *desired program model*, capturing the behaviour we assume the user wants to have for the object program.

For example, consider the Java program below and assume that the user wants to swap the names of the totalValue instance variable and the total method parameter. In the following, we have labelled important declarations and variable references with $[\cdot]^n$.

```
class Receipt {
  [int totalValue]¹;
  void setTotal([int total]²) {
    [totalValue]³ = [total]⁴;
  }
}
```

A rename refactoring using atomic preconditions would disallow starting the transformation by renaming either totalValue to total or total to totalValue because in both cases the parameter (2) would capture the left-hand side of the assignment (3), possibly changing the behaviour of the program. This transformation *could* be accomplished via refactorings, albeit awkwardly, by first renaming one of the variables to a temporary name, renaming the other to the first name, and then renaming the temporary name to the second name, but this work-around requires three steps rather than two and forces the user to plan ahead when refactoring.

Program metamorphosis can perform the transformation safely in two steps using postconditions. It starts by creating the following program model, which captures the binding of variable uses to declaration (here $n \rightarrow m$ indicates that the variable used at n refers to the declaration at m).

Desired Name Model	3 → 1, 4 → 2

The user can then apply a rename step to change the total field declaration (1) to be called totalValue. Program metamorphosis now computes the model for the transformed program and compares it to the desired model, reporting any inconsistencies to the user. In this case, the rename has caused the reference on the left-hand side of the assignment (3) to be captured: it previously referred to the field (declaration 1) but now refers to the parameter (declaration 2).

```
class Receipt {
  [int totalValue]¹;
  void setTotal([int totalValue]²) {
    [totalValue]³ = [totalValue]⁴;
  }
}
```

Desired Name Model	3 → 1, 4 → 2
Current Name Model	3 → 2, 4 → 2
Inconsistencies	3 captured by 2

The program's behaviour has now been changed: calling setTotal() will no longer update the totalValue field. To ensure behaviour preservation, the user can either revert the rename transformation, or apply another rename step to rename the left-hand side totalValue (3) to total.

Our transformation steps have access to the desired program model, which helps them in transforming programs. We thus distinguish them from arbitrary transformations by referring to them as *PM steps*. In the case of rename, we use the mappings in the *desired model*, rather than the current model, to decide which occurrences of totalValue need to be changed. Thus renaming (3) to total updates both (3) and the field declaration (1), since (3) is mapped to (1) in the desired model, while leaving (2) unaffected:

```
class Receipt {
  [int total]¹;
  void setTotal([int totalValue]²) {
    [total]³ = [totalValue]⁴;
  }
}
```

Desired Name Model	3 → 1, 4 → 2
Current Name Model	3 → 1, 4 → 2
Inconsistencies	None

The current model now matches the desired model. We have achieved the desired transformation safely and naturally in only two steps, which would be impossible with a traditional '*Rename*' refactoring since atomic preconditions prohibit transforming through an intermediate stage that does not preserve behaviour.

Note that in our example, we not only observed whether the current and desired program models matched (≡), but also how they diverged. The inconsistencies we reported during the example are crucial for guiding the programmer to successfully transforming the program.

Figure 3.4 summarises our approach:

(1) We first calculate a model for the object program and save it as the *desired program model.* We call this the *desired* model since we ultimately want the transformed program to end up with the same model.

(2) When the user applies a PM step, we compare the desired program model with the calculated current program model, reporting any inconsistencies to the user. In this

way, our desired model takes a role equivalent to that of the high-level specification in program refinement (except for being less high-level). If there are inconsistencies between this desired and current model, the user may:

(a) *Revert* the previous step,

(b) *Continue* by applying another PM step,

(c) *Accept* any or all of the reported inconsistencies as behavioural change by updating the desired model to incorporate the change in behaviour. Thereby, the user may 'update the specification' during transformation.

(d) *Search* for recovery plans: therein, we search the space of all possible transformation sequences to find those that will satisfy the postcondition. Users can then pick which plans (if any) they want to enact. Such an approach falls within the realm of *AI Planning*; to be practical, it requires heuristics that can guide the planning process (Section 8.1).

The key benefit of program metamorphosis is that after the user has applied any particular PM step, the program's current behaviour may not match its original behaviour, but *the user can continue applying steps until it does*. In this way, PM allows the composition of simple, possibly non-behaviour preserving steps into a sequence that does, in its entirety, preserve behaviour.

3.3 The utility of safety guarantees

The precise nature of the properties captured in the program model and the inconsistency checking process determine how useful our safety guarantees are in practice. For example, consider a model that is identical to the program, and a consistency checker that issues an inconsistency whenever there is syntactic difference. This model is overly conservative and thereby issues many false inconsistencies (e.g., after indenting code or altering a comment). However, the model has perfect *recall*: it will identify any behavioural change in the program, as such change must be manifest somewhere in the source code.

Conversely, a program model might choose to capture no information at all and never issue an inconsistency, turning the program metamorphosis system into (in essence) a fancy text editor. Such a program model is overly optimistic, though it has perfect *precision*: it does not report a single false inconsistency.

In practice, program models and consistency checkers must choose some tradeoff between precision, recall, and computation time; in Section 6.2 we illustrate some examples for the range of choices that program metamorphosis system designers have.

However, for many program models there is one approach that allows us to refine the precision of our guarantees at minimal computational cost, as we describe in the next section.

3.4 Refining safety guarantees with external knowledge

The 'transform and accept' approach of program metamorphosis is sufficient for refactoring and some program evolution purposes, but it is too limiting for at least one of the tasks we set out to support, namely algorithmic optimisation. For example, assume that we want to replace calls to function f_slow with calls to another, faster, function, f_fast. Instead of accepting the change for each call site individually (which is little better than manual editing), we would

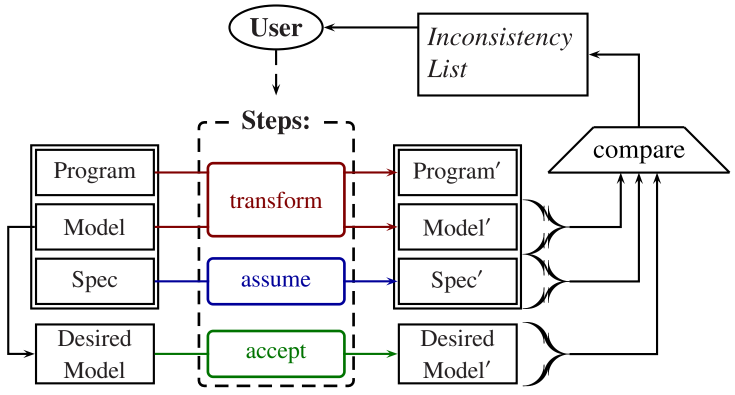


Figure 3.5: Program metamorphosis process with Assume step.

much rather tell the system once that our two functions ‘do the same thing’ (unless, of course, the system can already infer this information.)

In Figure 3.4, we show an extension to our earlier overview of basic program metamorphosis that facilitates this concept by adding a *spec*, or *axiomatic specification*, to the program and a new kind of step, *assume*, to the transformation process. This ‘spec’ is a collection of axiomatic information that the program metamorphosis system can use to eliminate false inconsistencies. In our example, the spec would encompass an axiom that tells us that f_slow and f_fast have identical behaviour, though the knowledge contained in a spec need not be equational in nature. For example, in Chapter 5 we describe *congruence specifications* as a form of specification that we have found particularly useful for algorithmic optimisation.

Note that the correctness of such specifications has considerable impact on the utility of our safety guarantees: ‘correct’ axioms (i.e., axioms that properly reflect the implementation) reduce the number of false positives, while incorrect axioms increase the number of false negatives. If, as we suggested, users may provide such specifications, a program metamorphosis system might ask or look for proof for each axiom, depending on its provenance:

(1) *Library axioms.* A program metamorphosis system may choose to ship with additional knowledge that will be embedded into the equational process. For example, arithmetic operations might be part of this knowledge base, allowing the system to reason that an expression g() + 0 is identical to g(). Such library axioms are thereby part of the system’s trusted base: if a library axiom is incorrect, the program metamorphosis system may miss inconsistencies due to no fault of the user.

(2) *Inline axioms.* Users may specify axioms as part of their programs, e.g. by using a special comment format. This is useful to inform the system e.g. about properties for custom datatypes and algorithms. A program metamorphosis system may or may not be able to show or at least test whether these axioms indeed reflect the implementation; alternatively, the system may issue a set of proof obligations from the axioms, the object language, and the object program and require or encourage an offline proof.

(3) *Assumptions.* Through the *Assume* step, users may add new axioms during transformation. These assumptions have the same ontological status as inline axioms.

3.5 Outline

We have now seen how our approach can tackle multi-step transformations (Challenge #1 from Section 2.4) and safe behaviour evolution (Challenge #2). With a suitable choice of program model and program transformation steps, a program metamorphosis system can exceed the expressive power of existing refactoring tools whilst giving near-perfect behaviour preservation guarantees. After reviewing related work in the next chapter, we give two extended examples to illustrate our approach's transformative power in Chapter 5, first by showing a refactoring on Java that has so far eluded all existing refactoring tools (to the best of our knowledge); this again illustrates our handling of Challenge #1. For the second example, we show how our approach can handle an algorithmic optimisation (Challenge #3), wherein we replace lists by vectors in an SML program.

We then consider again and in more detail how our program metamorphosis prototypes operate. Recall from our earlier description that a program metamorphosis system implementation requires at least the following components:

- a *program* representation,
- a *program model* representation,
- a *program model analysis* (properties) that computes program models from programs,
- a *consistency checker* (≡) that compares two program models and generates inconsistency sets,
- a set of *PM steps* that simultaneously update program and program model,
- appropriate infrastructure to glue these parts together and integrate them with a user interface.

While the detailed structure of each of these components depends on the needs of the concrete program metamorphosis system, many of the ideas are shared. Chapter 6 describes the general structure of the program model as well as giving a survey of the program models, program model analyses, and consistency checkers that we have used. In particular, this section explains how we can address Challenge #4, the separate handling of side effects and value results.

In Chapter 7 we then discuss general mechanisms for constructing program metamorphosis tools and describe a prototype program metamorphosis tool generator that we have implemented.

Chapter 8 then summarises our three prototypes, their program models and transformative power, while Chapter 9 evaluates them, both by describing their transformative power and by discussing the strength of the behaviour-preservation guarantees that our approach offers. In particular, we describe how one of our prototypes offers near-absolute behaviour preservation (Challenge #5) without being overly conservative about the equivalences it permits.

Chapter 10 then lists some new questions and challenges that our work has opened up, and Chapter 11 summarises our contributions.

Chapter 4

Related work

There is a considerable body of related work both on transformation systems (particularly refactoring systems) and techniques that we can base such systems on.

We thus separate related work into a number of categories, beginning with program transformation tools (Section 4.1).

For program metamorphosis, we find it necessary to detect and compare behaviour between two versions of a program. This is in principle similar to problems that Kim's work on structural change inference faces [KNG07]. Kim compared different program revisions, based on subversion records, and constructed more abstract change descriptions from the data she mined. Her challenge is different from ours, though, as we have more detailed information about the program before and after each individual transformation available. We thus turn our attention towards mechanisms for building (Section 4.2) and computing program analyses (Section 4.3) to construct our program models, before discussing other contexts that deal with behaviour preservation in programming languages (Section 4.4).

4.1 Program transformation

Program transformation is a technique used in a number of engineering and re-engineering tasks. Such transformations may be free-form or in the shape of 'equivalence transformations'. Systems that perform such a notion of 'equivalence transformations' on programs have now been the subject of research for about thirty years. Such systems have been designed with different goals in mind, such as increasing the quality of code as a document [Ars79] or to improve code performance [Lov77]. We consider these two particular goals as the central point of distinction between refactoring and program refinement: The goal of refactoring is to increase the quality of a program as a document, whereas program refinement strives to improve the dynamic behaviour of a program.

Applying this distinction, we begin this section with a discussion of refactoring systems, followed by program refinement, and concluding with general-purpose program transformation systems.

4.1.1 Refactoring

Refactoring systems are systems that explicitly support the refactoring process, usually by providing automatic, behaviour-preserving transformations (or some approximation of behaviour-preserving transformations, cf. Section 2.2).

Some of the more prominent refactoring systems are Eclipse [SDF$^+$03] and IntelliJ [Sof]. HaRe [TR01a] appears to be the only refactoring system presently available for functional languages. For logic languages, Schrijvers et al. [SSD01] describe a refactoring tool for Prolog. Roberts et al. [RBJ97] describe a rather well-known system for Smalltalk.

A number of approaches have been attempted to verify refactorings; e.g., Mens et al. use Graph Theory [MEDJ05, MDJ02, ES95] to check for correctness in object-oriented programs. Garrido and Meseguer [GM06] use an algebraic approach, as do Borba et al. [BSCC04], though the former also use explicitly executable specifications. Li and Thompson [LT05] use lambda graphs as their underlying formalism [AB02]. Finally, Kataoka et al. test for dynamically inferred specifications to ensure correctness [KEGN01].

Noteworthy dissertations in the area are those of Griswold [Gri92, GN90], who describes a program transformation system for LISP, and Opdyke [Opd92], who originally coined the term ‘refactoring’.

While there has been prior work on composing atomic refactorings to construct bigger refactorings (as investigated by Kniesel and Koch [KK04]), we are not aware of any systems that allow temporary invalidations, or even just dynamic composition.

Recent work by Schäfer, Ekman and de Moor [SEdM08b, SVEdM09] and independently by Steimann [ST09] suggests that the direction in this field is turning towards a ‘transform, check, and abort or fix on failure’ approach, rather than the traditional ‘predict and transform’ approach suggested by Fowler’s preconditions [FBB$^+$99]. We find that this coincides with our observations on numerous levels.

Finally, Mens provides the most recent (2004) survey of the state of refactoring [MT04].

4.1.2 Program refinement

We use the term ‘program refinement’ to describe a general class of approaches towards transforming programs from abstract specifications down to efficient implementations. Within the Program Refinement community, the term ‘Program Transformation’ is more commonly used for this approach [BD77]; however, we find this term to be overloaded and therefore avoid it for these purposes.

Program refinement systems usually operate on a ‘Wide-Spectrum Language’, named thusly due to a wide linguistic range on the level of abstraction.

For example, the WSL used by the Munich Project CIP [BBM$^+$85, BEH$^+$87] uses, at its most abstract, algebraic specifications. During implementation, programmers can concretise the specification down to the level of imperative programs with array and pointer operations [BBM$^+$85]. WSLs may be restricted arbitrarily to simplify the task of program refinement, e.g. the CIP WSL does not allow aliasing of reference parameters to functions.

Another example of such a system is Ward et al.’s FermaT system [WZ05]. FermaT is noteworthy in that it allows user-defined transformations, expressed in MetaWSL [WZ05]. Such MetaWSL programs are built from primitive AST node transformations, and may again be subjected to program refinement, since MetaWSL is an extension of the FermaT WSL. It is unclear whether this approach has any implications on the ease of proving the behaviour preservation property of MetaWSL programs.

Except as noted above, program refinement systems normally utilise a built-in set of language-specific axioms [Bac03] as transformation rules.

In the HOPS System [Kah99], axioms and programs are instead represented as graphs, though the general approach is similar. However, through its particular use of graphs, HOPS makes types explicit in its program representation, and graph transformations are applied to terms and types simultaneously.

An interesting example of 'vertical' program refinement can be found in the VISTA system [ZCW$^+$02], which allows user-driven changes to generated assembly code. The system then attempts to validate any concrete changes by means of an automatic (and therefore incomplete) mechanism.

4.1.3 Other source-to-source transformation systems

General-purpose source-to-source transformation tools allow the formalisation of a broad class of program transformations. Below, we describe several such systems.

Cordy's TXL system [CDMS02, Cor06] is a conditional pattern re-writing engine with a number of nonstandard features, mostly aimed at easy experimentation for language design. TXL supports fixed-point computations and has been used for source code analysis of very large code bases, re-writing programs into programs annotated with analysis results.

Visser et al.'s Stratego language and XT toolset [Vis04, BKVV06] is a program transformation system that allows annotations to be added to AST fragments during traversal. Such annotations can be structured data, and thereby allow information to be embedded similarly to our relations. However, their embedded information is tied directly to AST fragments and not easily accessible globally; furthermore, their property derivation strategies do not separate program model facts from derived properties, causing the combination of transformations and analysis to be "problematic" [Vis04].

Appeltauer and Kniesel's GenTL system for conditional transformations [AK08] is an instance of a conditional transformation system, i.e. a system that implements program transformations and predicates together. While this is similar in style to refactorings, they do not appear to explore the more involved program analyses typically needed for refactoring.

Boshernitsan and Graham's iXj system [BG04] is another example of a source-to-source transformation system. Their central concern is the construction of a user interface for convenient pattern generation, which we consider to be beyond our scope.

A central concern of such transformation tools is the 'strategic' combination of individual small-scale rewritings [Vis99,LV97]. Similarly, we expect that we will find it useful to compose our small-scale transformations into larger transformations on the UI level, which we expect to allow us to recover the traditional notion of refactorings.

An up-to-date list of program transformation tools of various kinds can be found online, in the Program Transformation wiki[1] .

Compiler construction tools form a generalisation of general-purpose program transformation tools, in that the output is not restricted to being a program. Examples of such tools are ASF+SDF [vdBvDH$^+$01], CENTAUR [BCD$^+$88], the Synthesiser Generator [RT84], LISA [HPM$^+$05] and the PSG system [BS86]. The latter system is noteworthy in that it generates a language interpreter based on a translation into the lambda calculus, which the authors refer to as a 'denotational specification[2] .' Heering and Klint [HK99] provide an overview of these and similar systems. Heering and Klint note that many language-specific tools can be

[1] http://www.program-transformation.org

[2] A more accurate name might be a 'translational specification', since the lambda calculus allows objects with the same meaning to be represented differently. However, the denotational semantics of the lambda calculus are well-understood, so this is a mere nitpick.

generated by compiler generation tools, but note that ‘maintenance’ tools are still missing from this set of language-specific tools.

Combined design recovery/program transformation tools are used for re-engineering tasks, during which programs are translated into other programming languages or subjected to major design changes while behaviour is preserved to some extent.

One example of an entire class of such systems is the Software Refinery [vD97,MNB$^+$94], which is a system for developing software re-engineering tools, though the system appears not to offer any guarantees about the behaviour of the generated tools. Re-engineering tasks must deal with an entire class of problems that we disregard in our work, such as language dialects [vD97] or invalid source code [MCC$^+$01].

4.2 Static program semantics

Static program semantics capture overall program well-formedness as well as some amount of program behaviour (such as public interfaces). Nielson et al. [NNH99] present a summary of the most popular techniques for computing such static semantics. Analyses that are of particular interest in our context are attribute grammar analyses performed for program analysis (Section 4.1.3), effect analyses (Section 4.3.2), but also type inference mechanisms, which we must support fully in order to support Standard ML.

Pottier and Rémy [PR05] present a mechanism for inferring type constraints for SML in separation of type checking, and using a constraint solving/checking mechanism to guarantee type correctness. This mechanism is similar to possible Datalog-based analyses that we employed and explored in our initial prototype.

4.2.1 Logic programming in program analysis

Cohen et al. describe JTL [CGM06], a general-purpose query system and language for inspecting Java programs by properties (including some semantic properties). Their approach is equal to ours, in that they use Datalog for describing and querying program properties. Unfortunately, the authors make contradictory statements regarding whether their system is bottom-up (forward-chaining) or top-down (backward-chaining, like ours).

Numerous approaches to speeding up Datalog queries have been proposed in the literature. Ross [Ros96] provides a quick overview of several techniques, including his own contribution of tail recursion elimination; Bravenboer and Smaragdakis [BS09] show additional techniques for state of the art Datalog query systems (demonstrated on a family of points-to analyses).

4.3 Dynamic program semantics

Dynamic program semantics describe program behaviour during program execution. There are various forms of such semantics that have been employed to describe programming languages in the past and even to generate interpreters directly from specifications (cf. our notes on compiler construction tools in Section 4.1.3).

We borrow some of the ideas from operational semantics and effect handling as basis and inspiration for some of our safety guarantees, and for our soundness proofs.

4.3.1 Operational semantics

Operational Semantics (as suggested by Plotkin [Plo81] and later refined, as 'Natural Semantics', by Kahn [Kah87]) are schemata for steps of evaluation. Such schemata are serialised in the form of inference rules, which are meant to express a single 'step' of evaluation. Operational semantics are particularly relevant for us, since the definition of Standard ML [MTHM97] is given in such a formalism (aided by a number of auxiliary mechanisms). Since operational semantics directly describe an evaluation model, they can be applied directly to implement an interpreter; for example, the HaMLet System [3] directly implements the SML97 specification. On the other side of the application of Operational Semantics, Pop and Fritzson [PF05] have developed a debugger specifically for natural semantics.

Considering the official specification of SML as being given by operational rules, it would seem to be natural to base our proof mechanisms directly on the operational SML semantics. This approach has been attempted previously in the EML project [KST97, Kah, KS98]. Kahrs et al. note in their 'Gentle Introduction' to the SML dialect Extended ML [KST97], "The size and complexity of the semantics is such that fully formal use of it, e.g. to prove correctness of an optimising transformation, would be quite a difficult task," referring explicitly to the SML90 semantics. Nonetheless, they pursued this very approach, though it eventually turned out that proofs on operational semantics required 'higher-order inference rules' (inference rules whose premises contain inference rules). To the best of our knowledge, the project was not continued towards the development of a proof assistant, and we believe that this is not least due to the complexity mentioned by Kahrs et al. in the above quote.

Kahrs et al. explicitly point out 'denotational or algebraic-style semantics' as an alternative, which we shall consider next.

4.3.2 Effects

Effect systems (a variant of type systems) can be used to model and infer side effects in programs, cf. work by Lucassen and Gifford [LG88], and by Talpin and Jouvelot [TJ92]. We may adopt some variation of their systems for our effect inference mechanisms.

4.3.2.1 Monads and arrows

Monads [Mog89, Wad90a, Wad92, JW93, BHM00, Hug00] are a mechanism for achieving a number of goals at the same time, and specifically two goals of relevance to us: first, monads 'type effects,' in that types with monads and types without monads distinguish between (possibly) effectful and (guaranteed) pure functions, respectively; this allows us to enforce sequencing on effectful operations without impairing the utility of effect-free ones. Secondly, monads provide a denotational explanation for effects: By operating as type constructors, monads describe how types can be lifted into domains that combine regular values with effects. For that reason, the purely functional programming language Haskell [has97] bases its notion of side effects entirely on monads.

4.3.3 Systems for specifying semantics

There are several systems that use dynamic semantics specifications for programming language development; to our understanding, these semantics are only used for deriving lan-

[3] http://www.ps.uni-sb.de/hamlet/

guage interpreters. Our discussion of Compiler Construction Tools in Section 4.1.3 lists several such systems (specifically ASF+SDF [vdBvDH^{+}01] and the PSG system [BS86]); other systems of note are Goguen's OBJ family of languages [Gog00] and the SEMANOL system [ABB76].

4.3.4 Proving properties about meta-programming

Proof theory offers some additional ideas that we can use to check whether two programs have the same behaviour. Specifically, Gabbay and Pitt's work on variable binding [GP01] as well as deBrujin's work on indexed variables [dB72] may be of interest to abstract over variable names when comparing programs. In our work, we found consistent renaming to be sufficiently elegant, but we expect that future work may push greater demands on concise modelling of program behaviour.

4.4 Preserving behaviour in other contexts

The problem of (full or partial) behaviour preservation is of relevance to a number of other areas, specifically within the realm of Programming Languages. In particular, numerous approaches have been attempted to guarantee that compiler optimisations preserve program behaviour; perhaps most prominent are Lerner, Millstein et al.'s Cobalt [LMC03, LMC05] and Rubidium [LMRC05] systems.

However, the preservation of behaviour permeates all aspects of programming languages and their use; for example, Chase [Cha88] noted that certain kinds of garbage collection may alter program behaviour by running out of memory sooner than others.

Chapter 5

Transformative power

So far, we have only considered a single renaming example to illustrate our approach's utility. In this chapter, we demonstrate our approach's transformative power by considering two more involved examples.

Note that transforming programs is a relatively straightforward task, so we will place particular attention on the mechanisms we use for behaviour preservation, i.e., our program models (we discuss the strength of our behaviour preservation guarantees more holistically in Chapter 9).

First, we show how program metamorphosis can facilitate a refactoring that has so far eluded all existing refactoring tools (to our knowledge) in Section 5.1, and demonstrate that a name model alone is sufficient to give a limited amount of safety. In Section 5.2 we then show how program metamorphosis can facilitate algorithmic optimisation with a more sophisticated program model.

5.1 Teasing apart inheritance

Fowler [FBB+99] lists a "big refactoring" called '*Tease Apart Inheritance*', for cleaning up class hierarchies that do not clearly separate responsibilities. This refactoring is hard to fully support with traditional refactoring approaches but useful for showcasing some of the strengths of our approach. For example, consider Figure 5.1(a), where we have a class NetworkServer with subclasses TCPChatServer and UDPDataServer: here we have hardwired application protocols (Chat/Data) to transport protocols (TCP/UDP).

Assume that the method listen in both TCPChatServer and UDPDataServer has the following form:

```
void listen()
{
  ...
  process();
  ...
}
```

Since our classes TCPChatServer and UDPDataServer combine features that should be handled orthogonally, we wish to tease them apart, by moving the different implementations of method process into a separate inheritance hierarchy. Figure 5.1(c) illustrates this idea:

What we want is a universal (abstract) NetListener class that handles the TCP and UDP protocols in sub classes TCPListen and UDPListen. This class NetListener has a field s that aggregates an abstract Server class, with two concrete implementations, ChatServer and DataServer. Here, NetListener and its subclasses provide the method listen and Server and its subclasses provide the process method.

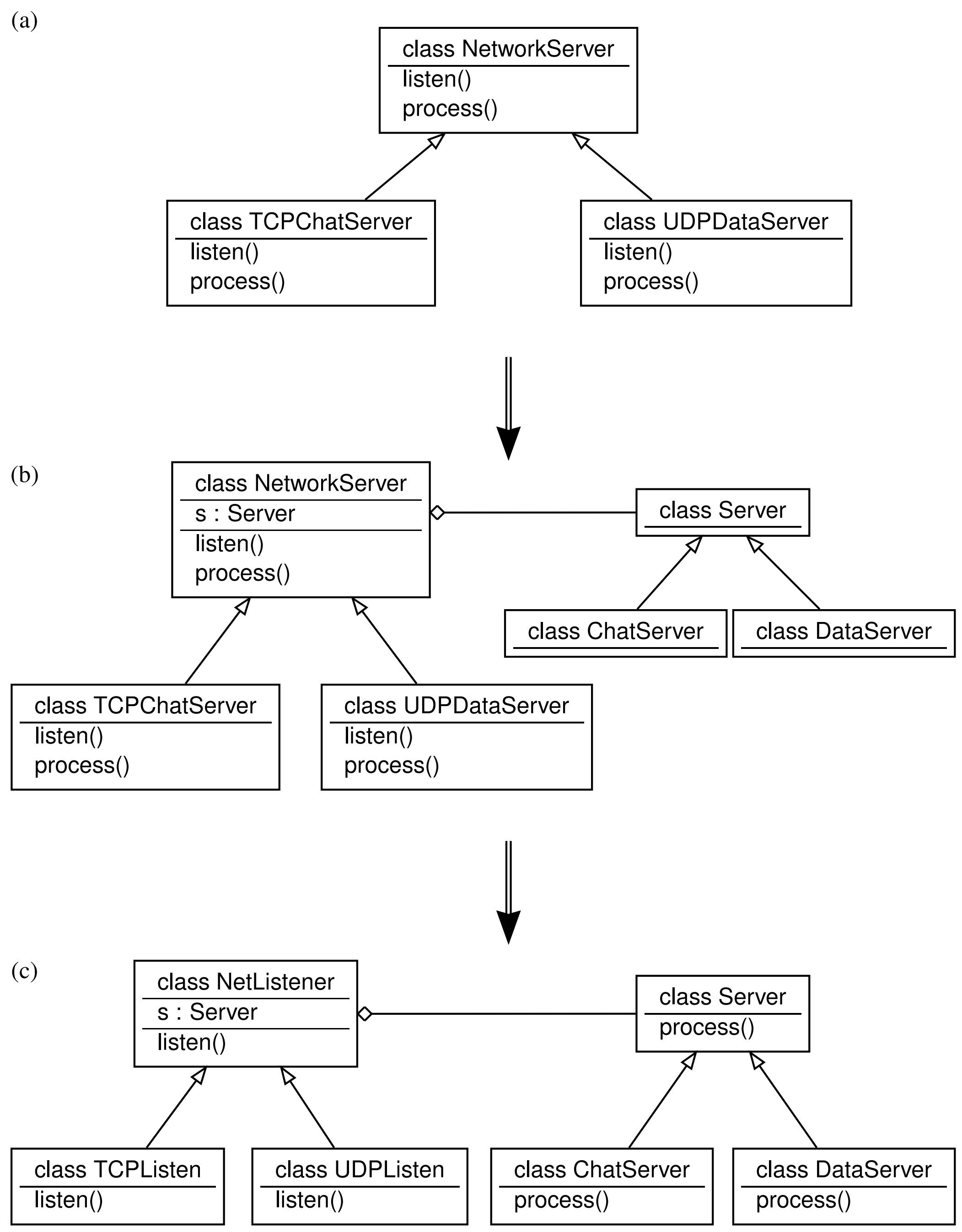


Figure 5.1: An example for the ‘Tease Apart Inheritance’ refactoring.

5.1.1 Transformation

With program metamorphosis, we can do this transformation straightforwardly:

Below is one possible way to transform the program (recall that program metamorphosis is generally lenient about the order of steps):

(1) Introduce new class (Server)

(2) Introduce new field (s) into NetworkServer

(3) Introduce new subclass (ChatServer <: Server)

(4) Introduce new subclass (DataServer <: Server)

We now have two parallel hierarchies and aggregation, but all of our functionality is still in the old hierarchy (NetworkServer and its subclasses), as in Figure 5.1(b).

We now move the functionality over:

(5) Move method (process()): NetworkServer → Server

(6) Move method (process()): TCPChatServer → ChatServer

(7) Move method (process()): UDPDataServer → DataServer

For cleanup, we now rename our 'old' classes as in Figure 5.1(c), to emphasise that they have been re-purposed:

(8) Rename NetworkServer → NetListener

(9) Rename TCPChatServer → TCPListen

(10) Rename UDPDataServer → UDPListen

Note that this is not quite enough, as our methods to listen() in TCPListen and UDPListen still contain calls to process(), which is no longer defined in these classes, meaning that the program is invalid after these steps. We can fix this by explicitly delegating this call to the aggregate:

(11) Delegate process() → s.process() in TCPListen

(12) Delegate process() → s.process() in UDPListen

The resultant method bodies for listen() would then look as follows:

```
void listen()
{
  ...
  s.process();
  ...
}
```

One of our prototypes, the Java prototype (Section 8.2) provides most of the necessary transformation facilities for what we just described; the only missing parts are adding new (empty) classes and subclasses and new fields — all of which are trivial and have minimal (if any) impact on program behaviour.

5.1.2 Program model and behaviour preservation

While the new program is now well-formed, it does not have the same behaviour as the previous program for two reasons:

- Globally speaking, we have not initialised s.
- Locally speaking, our methods listen no longer know whether their calls to s.process() will be invoking DataServer.process() or ChatServer.process().

(plus a number of reflection / public API change reasons that we ignore here for brevity.)

With a straightforward name model, such as the one that our Java prototype uses, we expect the system to give us inconsistencies for s.process(), since we use the static type of s to determine which process we are calling. These inconsistencies are accurate, in that they correctly identified a potential behavioural change: depending on how we initialise s, we may still have coupled TCP to the chat server and UDP to the data server, or not — this behavioural change is precisely what the refactoring intended.

In our prototype (which functions precisely in this fashion) we can address these inconsistencies easily by accepting them as a potential behavioural change.

Of course this still does not guarantee that s will be initialised correctly. While our Java prototype does not support this idea at present, we can make this particular refactoring safer by introducing a *parametric name model* (Section 6.3.4).

Note that we are not aware of any way to implement the above transformation directly using only traditional refactorings, and no refactoring engine we have experimented with supports '*Tease Apart Inheritance*' directly. We expect that a tool that wanted to support this refactoring in the traditional one-step form would have to provide a complex, special-purpose interface with highly involved preconditions.

5.2 Algorithmic optimisation

Next we turn our attention towards Challenge #3: replacing algorithms and datatypes by equivalent algorithms and datatypes.

Consider Figure 5.2(a). This code fragment defines a function read_file that loads a list of data and analyses them using the function analyse, returning both the analysis result and the data list. Function analyse in turn defines a helper function get: int → data to map list offsets to data.

Observe that get in Figure 5.2(a) uses List.nth, an $O(n)$ operation. This implementation choice may be prohibitively slow if dl is large. If we convert dl into a vector, element access time becomes $O(1)$. Figure 5.2(b) shows our program after this optimisation. We have introduced dv, a vectorised copy of our list dl. We now pass dv to analyse, which in turn uses Vector.sub, the $O(1)$ vector access method. If analyse calls get frequently, this change may speed up the program noticeably.

As in our initial example, we first consider the necessary transformations and then turn our attention to the program model, to understand how we can perform this transformation while ensuring behaviour preservation.

Figure 5.2: Algorithmic optimisation: replacing lists by vectors.

(a)

```
fun analyse (list : data list) : result =
    let fun get (i : int) = List.nth (list, i)
   ...
fun read_file (file : file) : result × data list =
    let val dl : data list = read (file)
    in (analyse (rl), rl)
    end
```

(b)

```
fun analyse (v : data vector) : result =
    let fun get (i : int) = Vector.sub (v, i)
   ...
fun read_file (file : file) : result × data list =
    let val dl : data list = read (file)
    val dv : data vector = vector (dl)
    in (analyse (dv), dl)
    end
```

5.2.1 Transformation

How we can get from Figure 5.2(a) to (b) depends again on the specific transformations that our program metamorphosis system provides. Below we list one path that captures how we would implement this transformation in our Standard ML prototype (Section 8.3):

(1) *Introduce definition*: Introduce a new definition, rv, bound to the expression 'vector (rl)'. Note that in SML, this transformation means that we might now raise a 'Size' exception, if rl is too long.

(2) *Replace subexpression*: We must replace rl by rv in the application of analyse. This change leaves the program ill-typed until analyse is also updated.

(3) *Replace subexpression* again: Once we update List.nth to Vector.sub in get, the program is well-typed and behaviourally equivalent to the original.

In our prototype (as in real SML) we must deal with the possibility of this algorithmic optimisation raising an exception. In our prototype, this is an 'accept' step:

(4) *Accept possible change*: The exception 'Size,' if raised, might alter the behaviour of the program. We must therefore explicitly accept this possible behavioural change.

For simplicity, we will ignore side effects in the rest of this chapter, but we will return to them when we discuss the full catalogue of our program models in the next chapter.

5.2.2 Program model and behaviour preservation

Conceptually, a program metamorphosis system cannot know about all the equivalences that exist between algorithms and datatypes. One reason is that we can always come up with a new algorithm that does the same thing as an existing algorithm, and we cannot expect the program metamorphosis system to detect this in general [Tur36]. Another is that even between existing algorithms and datatypes there may be unexpected but useful correspondences — if we try to enumerate all the 'interesting correspondences' between algorithms and datatypes in even just a moderately sized library, we would be hard-pressed to argue that we have enumerated all possibilities that might ever be of use.

Thus, we find ourselves forced to make the class of equivalences open-ended and user-extensible, with all the consequences we discussed in Section 3.4. To understand the behaviour preservation guarantees of our program model (Section 5.2.3) in the context of such specifications, we therefore find it helpful to reflect on a specification mechanism that captures these semantics — our SML prototype uses a specification language, which we describe in Section 5.2.4. Afterwards we sketch our program model in some more detail, along with a notion of congruence casts (Section 5.2.5). We then describe how we can scale congruence casts beyond local analysis to support our example in the above (Section 5.3).

5.2.3 Term model

A *term program model* is a program model that represents each expression in the program by a term such that the term describes the expression's computational result. In a pure language, a term model captures anything about the program that is semantically interesting, but in an impure language (such as SML) a term model will only capture part of the program semantics; we discuss how we extend 'simple' term models to Standard ML in Section 6.2.3.

For the rest of this chapter, the pure view suffices, as we will focus on examples that make sense without side effects. For the subset of SML that we consider here, the following (informal) grammar captures all of our terms:

$$t_1, t_2, \ldots ::= x \mid c \mid @(t_1, t_2) \mid \lambda x.t_1 \mid \langle t_1, \ldots, t_k \rangle$$

where x represents variables and c constants, such as literals, $@(t_1, t_2)$ represents function application, $\lambda x.t_1$ represents function abstraction, and $\langle t_1, \ldots, t_k \rangle$ represents tuples. In our examples we only construct tuples to pass them to library functions, so we need no tuple element projection or pattern matching operators.

For example, we might now represent the program

```
fn x => x + 1
```

as the term

$$\lambda x.@(+, \langle x, 1 \rangle)$$

using notation

$$\text{fn x => x + 1} : \lambda x.@(+, \langle x, 1 \rangle)$$

Note that this is an entirely straightforward isomorphic representation with no twist or depth. We introduce this notation here since we will later extend it to construct somewhat less uninspired term model languages.

In a full term model, we would then map each identifier to its representing term, and identify the 'toplevel' term that describes the program as a whole. If we assume that all names are unique, this is (as we noted) enough to capture the entire behaviour of the program.

In practice, function and value names are not all unique and not all globally visible, so we augment the term model with a name model to ensure that the program remains well-formed. We can thus consider the term model in isolation from such concerns.

5.2.4 Term congruence specifications

Armed with knowledge about our terms, we can turn our attention to the question of when two term models are equal. The easiest case is if all terms in the model are identical (as might be the case after a renaming transformation — recall that the term model operates in a world in which all names are unique). For anything more involved, we use term equivalences, of which almost all stem from specifications in our specification language (we list the exceptions later.)

Many term equivalences, including ones used in optimisation, are straightforward to express. For example,

```
axiom x + 0 = x
```

(here given in our specification language) gives a simple example: for any x, $x + 0$ is the same as x. Another example would be the following:

```
let fun f (x) = ...
    fun g (x) = ...
    (*# axiom x > 0⟹f(x) = g(x) #*)
in f end
```

This is an inline specification forming an SML comment. Our system identifies it as a specification by the surrounding '#' marks. This axiom will treat the local definitions of f and g as interchangeable, as long as the parameter passed to them is greater than zero. This is an example both of a local axiom and of a conditional one.

All of the above are examples of *equational* axioms: they tell us that two terms (with any number of free variables) are identical. Equational axioms are sufficient for many algorithmic optimisations, but insufficient in many cases that involve datatypes.

Consider again a migration from lists to vectors. If we generate a vector from a list using the vector operation, what we build is not something that is *equal* to a list, but merely similar in behaviour — *congruent*, according to some user-defined notion of congruence. We cannot use the same operations on lists that we use on vectors, but if we use 'corresponding' operations on 'related' lists and vectors, we get the same result:

```
val l = [1, 2, 3]  (* simple list-and-vector example *)
val v = vector (l)
val true = (List.nth (l, 1) = Vector.sub (v, 1))
```

Here, we first create a literal list l, then construct a vector v from that list. Subscripting the first element of the list and the vector, as we do in the last line, should then yield the same result, since l and v are closely related.

We can express this with congruences in our specification language, as follows:

```
(*A1*) axiom l ≡ vector (l)
```

Here, Axiom A1 states that, for any list *l*, computing 'vector(*l*)' yields a vector that is *congruent* to *l*. This doesn't tell us very much yet, since we haven't specified what this particular congruence means.

Intuitively, we want this congruence to mean that the list and the vector have the same elements in the same places. Our Axiom A1 establishes this property, but how can we exploit it?

The implication in Axiom A2 makes this clear:

```
(*A2*) axiom l ≡ v⟹ List.nth (l, i) = Vector.sub (v, i)
```

If we have a congruent list and a congruent vector, then subscripting the same element from them will yield the same result (right-hand side).

These are the two most basic uses of congruences. There are other possible uses, most prominently preservation axioms such as

```
(*A3*) axiom l ≡ v⟹ List.map (f) (l) ≡ Vector.map (f) (v)
```

which tells us that modifying a congruent list and a congruent vector in the same way, namely by applying the same function f, preserves the congruence.

Using the relation (≡) we can now straightforwardly show our simple list-and-vector example correct: because v = vector (l), we can derive that l ≡ v by axiom A1; from that, we get that List.nth(l, 1) = Vector.sub(v, 1) by axiom A2.

5.2.5 Inferring congruences

This last proof of equivalence used knowledge from the entire program. In general, this technique is impractical: not only is term rewriting that encompasses the entirety of a program unlikely to scale, it also provides little useful information to the user when it fails.

Consider again our algorithmic optimisation example from Figure 5.2: to see that replacing List.nth by Vector.sub in function get makes sense, we must know that all possible

$$\begin{array}{rcl} i & ::= & 0, 1, -1, 2, -2, \ldots \\ t_1, t_2 & ::= & x \mid c \mid @(t_1, t_2) \mid \lambda x.t_1 \mid \diamondminus^{i} t_1 \end{array}$$

$$\begin{array}{rcll} @(\lambda x.t_1, t_2) & \Rightarrow & t_1[x \backslash t_2] & (\beta_{cc}) \\ \diamondminus^{0} t & \Rightarrow & t & (\diamondminus_0) \\ \diamondminus^{i}(\diamondminus^{j}(t)) & \Rightarrow & \diamondminus^{i+j}(t) & (\diamondminus_1) \end{array}$$

Figure 5.3: λ_{cc}: syntax and reduction rules. Confluence follows from the standard theorems about the lambda calculus and the associativity of addition on integers.

parameters v of analyse are vectors that used to be lists; to reason about this with term rewriting, we would need a deeply context-sensitive analysis — this is impractical and we need a different solution.

We remedy this problem through a notion of *congruence casts*, constructs in our term language that localise the impact of a congruence. Figure 5.3 describes a variation over the term language we discussed earlier. We omit tuples here for brevity, but instead introduce *congruence cast operators*, notation $\diamondminus^{i}$. To emphasise that the congruence cast operator has a semantics, we add reduction rules and call the resultant language λ_{cc}, the *lambda calculus with congruence casts*.

Before we try to understand the above reduction rules, especially those pertaining to the congruence cast operator, let us first consider where and how we would use such congruence casts in our running example from Figure 5.2. Here we migrated a program from lists to vectors. Figure 5.4 shows what the program might look like with congruence casts inserted to *justify* the individual transformations we applied. Here, $\diamondminus^{1}$ represents a 'list-to-vector' cast, and $\diamondminus^{-1}$ represents a 'vector-to-list' cast. In practice, these congruence casts are on the term model, but since the term model is a homomorphic map of the program in this example, our annotations are individually sensible.

We infer congruence casts as a way to locally explain transformations. For example, if we transition from List.nth (v, i) to Vector.sub (v, i), axiom A2 tells us that this makes sense if and only if the 'new' v is a vector alternative to the old v, giving rise to a list-to-vector congruence cast.

We list this idea as Example #1 in Table 5.1. In that figure, we show what our system would infer to match up a certain term in the *Desired* term model to a new term in the *Current* model. We give the result of this matching in column *Annotated*. Example #2 then is analogous to explaining how we can pass dv instead of dl to analyse. In Example #3, we apply Example

```
fun analyse (v) =
    let fun get (i : int) = Vector.sub ($\diamondminus^{1}$ v, i)
    ...
fun analyse_file (file : file) =
    let val rl : data list = read (file)
        val rv : data vector = vector (rl)
    in analyse ($\diamondminus^{-1}$ rv)
    end
```

Figure 5.4: Congruence casts inserted after transformation in Figure 5.2.

Nr	#1	#2	#3
Desired	λ s. List.nth (s,0)	[1,2]	(λ s. List.nth (s, 0))[1,2]
Current	λ s. Vector.sub (s,0)	vector [1,2]	(λ s. Vector.sub (s,0))(vector [1,2])
Annotated	λ s. Vector.sub ($\diamondminus^1$(s),0)	$\diamondminus^{-1}$ (vector [1,2])	(λ s. Vector.sub (s,0))(vector [1,2])

Table 5.1: Explaining an updated current program model from the desired program model by adding congruence casts.

#1 to Example #2 and use the reduction rules of λ_{cc}. The two congruence casts cancel each other out in the resultant program:

$$\begin{array}{lll} & (\lambda s.\mathsf{Vector.sub}(\diamondminus^1(s),0))(\diamondminus^{-1}(\mathsf{vector}[1,2])) & (\beta_{cc}) \\ \Rightarrow & \mathsf{Vector.sub}(\diamondminus^1(\diamondminus^{-1}(\mathsf{vector}[1,2])),0) & (\diamondminus_1) \\ \Rightarrow & \mathsf{Vector.sub}(\diamondminus^0(\mathsf{vector}[1,2]),0) & (\diamondminus_0) \\ \Rightarrow & \mathsf{Vector.sub}(\mathsf{vector}[1,2],0) & (\beta_{cc}) \\ \Leftarrow & \lambda s.\mathsf{Vector.sub}(s,0)(\mathsf{vector}[1,2]) & \end{array}$$

When congruence casts cancel each other out, the resultant program pieces fit together neatly. After all, if we pack a vector-from-list transition around a list-from-vector transition, the result is again a vector. Conversely, if we have a lone congruence cast in our program and cannot eliminate it, our program must be ill-formed: two parts of the program provide or expect information that disagrees.

Thus, a program that can eliminate all of its congruence casts is well-formed, or *congruence-neutral*:

Definition 1. *A term t from λ_{cc} is congruence-neutral iff after normalisation to t', t' does not syntactically contain a congruence cast.*

The above definition means that no parameter — not even function parameters — to a constant may contain a congruence cast after normalisation.

5.3 Type-checking congruences

So far, the only benefit we have gained from our congruence casts is that we can modularise reasoning — but we still have to go through an expensive inlining step to eliminate them.

$$\begin{array}{rcl} i, j \in \mathbb{Z} & ::= & 0, 1, -1, 2, -2, \ldots \\ \alpha \in \mathit{CTyVar} & & \\ \tau, \tau' \in \mathit{CTy} & ::= & \alpha \mid i \boxplus \alpha \mid i \boxplus \tau \to \tau' \mid i \boxplus \bullet \end{array}$$

$$i \oplus \tau \stackrel{\Delta}{=} \begin{cases} i \boxplus \alpha & \iff \quad \tau = \alpha \\ i + j \boxplus \tau' & \iff \quad \tau = j + \tau' \end{cases}$$

Figure 5.5: λ_{cc}: types and congruence increment operator.

$$\frac{E\{x \mapsto \alpha\} \vdash t_1 : \tau \quad \alpha \text{ fresh}}{E \vdash \lambda x.t_1 : \alpha \to \tau}\ (\lambda) \qquad \frac{z \in \mathit{Var} \cup \mathit{Const}}{E \vdash z : E(z)}\ (\mathit{var})$$

$$\frac{\begin{array}{c} E \vdash t_2 : i \boxplus \tau \\ E \vdash t_1 : 0 \boxplus (i \boxplus \tau \to \tau') \end{array}}{E \vdash @(t_1, t_2) : \tau'}\ (\mathit{app}) \qquad \frac{E \vdash t_1 : \tau}{E \vdash \diamond^{i} t_1 : i \oplus \tau}\ (\diamond)$$

Figure 5.6: λ_{cc}: typing rules. Note that type inference for this type system is straightforwardly decidable.

However, there are other mechanisms for propagating the information that e.g. term t is a conversion from a vector to a list, and to check that such conversions cancel each other out. Specifically, we can rely on type inference. If we type each term with its *polarity* — basically the exponent of our congruence casts — we can type-check that all congruences would cancel each other out if we were to normalise the program.

Figure 5.5 lists our types. i are again polarities, and we annotate both function types ($\tau \to \tau'$) and atom types ($\bullet$) with a polarity via ($\boxplus$). We use the type $\bullet$ to construct the types of constants; to understand the other types, consider Figure 5.6:

This figure describes our typing rules. Function abstraction (rule λ) and constants and variables (rule *var*) are straightforward. Polarity addition (rule $\diamond$) types a term of the form $\diamond^{i}(t)$ with the type of t plus the polarity i, thereby adding up (and possibly cancelling out) the polarities of nested congruence casts. To sum up polarities, we use the ($\boxplus$) operation from Figure 5.5, which adds polarities (and lifts type variables α without polarities to type variables with polarities, if needed.) Finally, function application (rule *app*) is again mostly straightforward, except for one aspect.

Recall that we place no restrictions on what kinds of values users may define to be congruent with each other. Thus, a user might (for example) define a congruence between a function and a tuple containing the same function twice: $f \equiv \langle f, f \rangle$. If our user now replaces $\langle f, f \rangle$ by f, we would represent this as $\diamond^{1}(f)$ and type it $1 + (\tau \to \tau')$ (for some τ and τ'). Since this value is really a 'function that ought to be a tuple', we must not allow it to be applied to a parameter, since we do not know how to type a tuple that has been applied to a parameter.

We now have all mechanisms in place to show that Figure 5.4 is 'fully transformed' without requiring any inlining. First, consider analyse's get function. Here, Vector.sub takes two arguments, both of type $0 \boxplus \bullet$ (primitives with no polarity); consequently, we must type v : $-1 \boxplus \bullet$ (since this is the only type that gives us $0 \boxplus \bullet$ if we add 1). Therefore, we have

$$\mathsf{analyse} : (-1 \boxplus \bullet) \to \bullet$$

Conversely, in analyse's function application we then pass a parameter of type $(-1 \boxplus \bullet)$. Thus, the two types match up perfectly.

In practice, we now eliminate all congruence casts from the term model so that this model again precisely reflects the program.

We can formalise this anecdotal observation and prove that any program that is well-typed according to our type system allows us to tell whether a given term is congruence-neutral just from its type (Appendix C).

Chapter 6

Program models

The purpose of a program model is to capture program behaviour, permit us to identify changes in program behaviour, and facilitate the identification of program locations that are responsible for such behavioural inconsistencies.

In the last chapter we saw two examples of program models and their use, and an example of two program models (term model and name model) being combined to form a composite program model. There are many more possible program models, including several that we have explored.

In this chapter, we first explore a common design consideration among program models, namely how and where the models are attached to the program AST, and how they can localise behavioural changes in such a way that we can give meaningful inconsistency reports to the user (Section 6.1). Section 6.2 then lists a catalogue of all the program models we have implemented, followed by some program models that we expect to be of interest to future exploration, in Section 6.3. Based on our concrete patterns, we then describe a set of commonalities in Section 6.4 that we have found to capture commonalities between our concrete models. Section 6.5 then describes how the user may interact with program models.

6.1 Locations and behaviour

We use program models to identify behavioural change in the form of inconsistencies, but without knowing any program locations that *justify* each inconsistency we find it hard to aid users in resolving them. The most straightforward way to identify locations in inconsistencies is to partition behaviour into fragments, *model attributes*, and associate each attribute with a location. For example, in our term model the attributes were terms, and we associated each term with an AST location. Since inconsistencies are then caused by several model attributes disagreeing in some form or another, we can use the pertinent attribute locations to identify locations that justify the inconsistency.

6.1.1 Localising behaviour

We have observed three methods to identify behaviour with locations: *AST association*, *intermediate program model association*, and *label association*.

AST association. In this approach we associate each behavioural property directly with a location in the in-memory representation of the object program, typically an abstract syntax tree (AST). For example, we might associate the type of an expression with the AST node representing that expression.

As von Dincklage showed [vDD08], it is possible to automate the task of AST association if we express the computation of our properties *constructively* (i.e., avoid negated properties as much as possible). We sketch in Chapter 7 how this idea applies to program metamorphosis.

In AST association, we add model attributes directly to the AST, or to a map that we maintain in parallel.

Intermediate program model association. Some program analyses benefit from sharing information, such as type or name information. This information-sharing may be easier to accomplish with an intermediate program model (Section 6.4.2), an observation we again share with von Dincklage. Specifically, some analyses may be easiest to implement on a normalised representation (e.g. Section 6.2.3), such as A-Normal Form [FSDF93] or a basic blocks model, that eliminates some of the 'warts' of the real language. To facilitate behaviour localisation, each component of the intermediate model must remember the program locations that justify it. Other models that rely on this intermediate model, *child models*, can then obtain the 'proper' program location or locations from the intermediate model as needed. In terms of our previous definition, such child models may then have model attributes on the intermediate model, whereas the intermediate model components form model attributes of the AST (or of their parent intermediate model).

Label association. In this approach we detach behaviour from the subject program. A simple example of label association would be an *API model* (Section 6.3.1) that contains a list of names (and possibly types) of definitions that the program is expected to export, without tying each name to a program location.

With label association, we therefore cannot explain inconsistencies through program locations — instead we must explain them on a 'high level' that the programmer can understand, such as by saying 'this module no longer exports a method named `foo`.'

6.1.2 Localising change

Once we have localised behaviour, we can 'blame' any changes in behaviour to whichever location we recorded for the attribute. Often, there may be multiple locations to blame. For example, changing the type of a function may trigger a type error in all places that use this function; the blame here belongs equally to the function change and to the incorrect expectations at the call sites.

Heuristically, if we assume that the user acts rationally, the 'blamee' for such a change should not be the location that the user just modified (since blaming this location amounts to suggesting that the user undo their most recent change.) However, these heuristics become less reliable if users change multiple pieces of code at the same time; for that reason, our current implementations do not explore this idea.

Not all behavioural changes come about due to a direct change in the behaviour of one program location: some behavioural changes are due to deletions, introductions of new program components, or propagated changes. We look at each of these kinds of change in turn.

Obsolete program components. Consider deleting an unused function definition: if the function is not externally visible, this change will usually not be a behavioural change. However, if this program component used to be visible e.g. through reflection mechanisms, or if the definition of this component had a side effect (e.g., in an initialiser for an obsolete constant), then such a change may be significant. The precise means for reporting such change may vary by model: if a component in a sequence of effects is missing, the most useful location to report is the program location before or after the expected effect, but for a potential change in reflection behaviour, it would be more natural to map the change to the modified module and to any reflexive invocations that might be affected.

Fresh program components. Conversely, new program components might add behaviour or invalidate properties about the absence of certain behaviour. It is tempting (and easy) to always blame the new component for such changes, but again it may be more helpful to assign blame to other affected program components. For example, if a new method definition overrides a superclass method, developers may find it more helpful to be pointed to locations that will call the method in question.

Cascading change. Program components rarely stand alone, so changing one component may affect others. This effect may be very pronounced. For example, consider a definition of lists, where each list is defined as a unique value and in terms of its predecessor list:

```
val l0 = []       (* empty list *)
val l1 = 1 :: l0  (* prepend the integer 1 to l0 *)
val l2 = 2 :: l1
val l3 = 3 :: l2
```

Now, assume that we maintain a model that represents the ‘value’ of each of the list definitions. If we now change the first list, the values of all lists change!

A more subtle variation of this issue comes about with effect analysis. Assume that we preserve the effectfulness behaviour of programs in a functional language, and assume that our current program is pure. If we now change a single function from pure to impure, and if all other definitions depend on that function, then we have again altered the entire program.

While these two scenarios share a problem, the second issue is more involved since it is harder to isolate.

For our sequence of lists example, we can use *attribute isolation* to suppress the cascade. Attribute isolation is a guarantee that an inconsistency in one attribute will not trigger an inconsistency in another. In our example, this means that all later lists would store a reference to their predecessor lists, but not the contents of that list. Attribute isolation thereby has another advantage: it requires that we do not needlessly duplicate data. The downside of attribute isolation is that comparisons become more involved. In our example, we know that l3 = [3, 2, 1], but to see this we must recurse through three references. Nonetheless, we used this approach in several of our models.

Another disadvantage of attribute isolation is that it may make it too easy for users to miss accidental behavioural change. Assume that, in our example, the user were to update the definition of [] to [0]. Without attribute isolation, a suitable model would flag all of l0 through l3 as ‘inconsistent.’ With attribute isolation, only l0 would be flagged, and if the user were to accept this change, all inconsistencies would be gone. This may be problematic if the user fails to see the implied change to e.g. l3. There are two techniques to mitigate this problem: *Change consequence visualisation* and *change propagation*.

Change consequence visualisation highlights all dependent attributes, thereby giving the user a visual cue to how wide-ranging the effect of their changes is.

Change propagation is a different approach that affects the ‘accept’ step: if the user accepts a change, we copy the *old* definition (from the desired program model prior to the update) to all dependent attributes in the desired program model. In our running example, our user might accept the change of l0 = [0], but at that point the system will update the definition of l1 from l1 = 1 :: l0 to l1 = 1 :: [], thereby triggering an inconsistency, as we still have l1 = 1 :: l0 in the current program model and l0 now unfolds to [0].

The advantage of change consequence visualisation over change propagation is that users will not be forced to accept the consequences of a change repeatedly when they understand them. The advantage of change propagation is that users who do not understand the impact of a particular change can investigate this change in considerable detail.

None of our prototypes currently supports either approach, though we expect to experiment with change propagation as an *alternative* (i.e., not a replacement) to regular change-acceptance in the future.

6.2 Examples of program models

In the following, we give a survey of the program models we have implemented or postulated. For each model, we describe its *scope*, *analysis*, *representation*, and *inconsistency detection*. Here, the scope determines (informally) the aspects of program behaviour that the model captures.

Most of the models we present are from one of our three prototypes: either our Datalog prototype (which we more fully describe in Section 8.1), our Java prototype (Section 8.2), or our SML prototype (Section 8.3).

6.2.1 Name models

A *name model* maps each name (function name, variable name, class name, template name, signature name etc.) to an *identifier* that uniquely identifies it. While names may be ambiguous (we may have several function parameters with the name x), identifiers are not (each parameter x will have its own unique identifier).

We preserve these identifiers in the desired program model. After each program transformation, we can re-run name analysis to determine whether the old identifier structure matches the new one.

Name models detect name capture, out-of-scope references, and some forms of naming ambiguities. These models have the distinction of being the only model that is part of all of our prototype implementations, though with rather different implementations. Section 6.2.1.1 sketches the relational model, from our first or *Datalog* prototype (Section 8.1). We omit the full details of this model from here and present them in the context of our Datalog prototype, since they highlight a path we initially considered but later abandoned. Sections 6.2.1.2 and 6.2.1.3 then briefly describe our name models for the Java and SML prototypes, respectively.

6.2.1.1 Datalog name model

In our initial prototype, we used two separate name analyses: the SML of New Jersey name analysis provided by the SML/NJ frontend [App92] as the initial analysis, and a custom, Datalog [Ull89, WACL05] rule-based analysis.

From the initial analysis we extracted both the current and the desired program model as a family of facts. These facts took the form of relations that we maintained in parallel with the program AST; in terms of Datalog they formed extensional database relations.

We then inferred inconsistencies (such as name capture) and model errors (such as ‘name not in scope’) through derived ‘intensional database’ relations (in terms of Datalog). Each element in any of these relations was a model attribute, AST-associated through at least one location. We thus found it easy to associate inconsistencies and errors to program locations.

Ultimately, we found the approach of using two identical analyses in two separate languages — one to construct the program model, and one to check parts of it for internal consistency — to be needlessly complex. However, the use of Datalog may open up paths for future work (Section 8.1).

6.2.1.2 Java name model

Our Java prototype is based on the Eclipse IDE [1] . For name analysis, this prototype used Eclipse's built-in bindings mechanism to determine the declaration for each use of a name and store a mapping between names and declarations. Since Eclipse does not persist ASTs across transformations, we relied on a label association mechanism.

6.2.1.3 SML name model

Our full SML implementation relied on a name consistency checking mechanism that we generated with our attribute grammar-based program metamorphosis tool generator (Section 7). While logically similar to our Datalog approach, this mechanism used a more efficient implementation scheme and did not rely on two largely identical analyses.

For name capture, our name model used the model refinement pattern (Section 6.4.4) with the term model: if the user accepted name capture, our system would double-check to ensure that both the old and the new identifier had a matching term model.

6.2.2 Use/Def model

In our Java prototype, we mapped each identifier use to its most recent update or initialisation, constructing a Use/Def model. For example, in the program

```
1    int y, x = 0;
2    print(x);
3    x = 1;
4    y = 2;
5    print(x);
```

the use of x in line 2 would map to the initialisation in line 1, whereas the use in line 5 would map to the assignment in line 3. This allowed us to safely implement the '*Split Temporary*' refactoring [FBB+99], and to guarantee that we would detect the inconsistency caused by swapping lines 2 and 3 (whilst correctly allowing e.g. lines 3 and 4 to be swapped).

6.2.3 Term and effect models

Term models capture the *pure* aspect of a program's semantics, i.e., the computational result without side effects. Effect models capture the opposite, i.e., only the *impure* aspect, the program's side effects.

However, these two aspects interact in nontrivial ways, as we illustrate on the next pages, thereby completing Challenge #4. Before seeing the details, we can gain an early intuition about this idea from the monadic perspective: if we model an effectful language in an effect monad α E, the term model captures the pure aspect α and the impure aspect all else — but, evidently, any operations that interact between these two components need special treatment.

[1] http://eclipse.org

$$\begin{array}{llll}
e \in \mathit{Expr} & ::= c \mid x \mid e_1\ e_2 \mid \lambda x.e \mid \mathsf{ref}\ e \mid\ !e \mid e_1{:=}e_2 \mid \mathsf{raise} & \\
& \quad \mid e_1\ \mathsf{handle}\ e_2 \mid \mathsf{print}\ e \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \\
& \quad \mid \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 & \\
\overline{e} \in \mathit{Val} & ::= c \mid \mathsf{raise} \mid \lambda x.e \mid a & \text{(SiML values)} \\
c & ::= \mathsf{true} \mid \mathsf{false} \mid \mathbf{()} & \text{(Constants)} \\
a & & \text{(Memory addresses)} \\
x & & \text{(SiML variables)}
\end{array}$$

Figure 6.1: Simple ML syntax.

We implemented term and effect models in our most recent prototype, the SML program metamorphosis engine (Section 8.3). Since we defined our term and effect models in terms of Standard ML, we give them below fir a simplified variant of SML, Simple ML (SiML). SiML, while lacking many of SML's pattern matching, typing, and abstraction features, is entirely sufficient to express effectful computations.

6.2.3.1 Simple ML

Simple ML (SiML) includes the standard primitives of the lambda calculus, but also includes ref-value construction, reading and updating (ref, !, :=), exception raising and handling (raise, handle), and printing (print). Figure 6.1 gives the syntax of SiML.

6.2.3.2 Separating terms and effects

Recall that the purpose of these program models is to detect behavioural changes in a program and report them as inconsistencies. However, it is insufficient to just detect the change; we must also clearly describe the changes to the user so that the user can deal with the changes appropriately.

As we previously argued, this means that we must be able to localise inconsistencies. But of course we must also be able to explain the *kind* of inconsistency, i.e., explain *what* it is that the program might be 'doing differently' at those locations. For our term and effect models, we draw inspiration for this idea of 'doing something differently' from the monadic lambda calculus [Mog88], wherein we distinguish between *computational results* and *side effects*: if we say let x = *e* in . . . , then the computational result is whatever x contains in the end, while the side effects are anything else that might happen while we evaluate *e*: prints, exceptions, memory updates and so on. Our system tries to tease these two ideas apart by representing each in its own model: computational results in the term model (Section 6.2.4) and side effects in the effect model (Section 6.2.5).

To see what exactly these two models represent, how we compute them, and how they collaborate, we now introduce both through a series of examples.

6.2.4 Term model

Consider the following SiML program:

$$e_1 = \mathsf{if\ true\ then\ true\ else\ false}$$

This is a SiML program that always yields true by taking the first branch in a conditional. If we want to detect changes to the computation of e_1, it is clearly sufficient to focus on the computational result, since we have no side effects. This means that behaviour preservation is solely in the hands of our term model. To ensure that the term model can identify whether e_1 has changed, it needs all the necessary information to identify e_1; we therefore capture the above computation in our term model by copying it verbatim:

$$e_1 : \textbf{if}\ \mathsf{true}\ \textbf{then}\ \mathsf{true}\ \textbf{else}\ \mathsf{false}$$

Here, $e : t$ infers the describing term t for a SiML expression e. (Note that we use a boldface font for term model constructs.) Figure 6.2 gives the inference rules for our term model; we will explain the θ later in Section 6.2.5.2. Rule (*T-if*) for if and rule (*T-c*) for constants generated the above model.

If we now want to check whether a transformation has preserved the behaviour of a side-effect free SiML program, we can do so by comparing the term models before and after the transformation.

If, on the other hand, our program has side effects, our term model is not enough: the term inference rules only capture the bare minimum information about effects. For example, rules (*T-a*) and (*T-print*) 'filter out' assignments and print statements, respectively, since both evaluate to **()** in SiML. Thus, we cannot rely on the term model alone to detect whether or not a transformation has changed the behaviour of a program.

6.2.5 Effect model

Our effect model captures all side effects of an expression in the order in which the effects occur. For example, in a program that prints first true and then false, the effect model must remember that we print these two values, and that we print them in that particular order.

Notationally, we put these effects into order through a sequencing operator, $\triangleright$. In the above printing example, our model is

$$(\mathsf{true} : \mathtt{io}) \triangleright (\mathsf{false} : \mathtt{io})$$

Here, '`io`' stands for the kind of effect that we are observing (printing) and 'true' and 'false' tell us what we are printing.

To see how we arrive at such sequences or *effect traces*, consider Figure 6.3, which lists our effect trace inference rules. Again, we can ignore the left-hand side of the turnstile (E and θ) for now; we will return to them later. The right-hand sides of our judgements have the structure $e : \eta$, with e a SiML expression and η an effect trace. In the conclusion of rule (*E-print*), which handles prints as in our above example, we see the judgement print $e : \eta \triangleright (t : \mathtt{io})$, which tells us that the effect trace for print e is $\eta \triangleright (t : \mathtt{io})$. The η represents the side effects when we evaluate e and $(t : \mathtt{io})$ represent the print itself; t is the computational result of evaluating e. The operator $\triangleright$ indicates sequencing; the effect to its left occurs before the effect to its right. In rule (*E-print*), we determine η through recursion on e, and t by relying on our term model analysis.

On top of printing, our effect model must consider two other kinds of effects that SiML uses and that our term model cannot fully capture: ref-values with assignments (Section 6.2.5.1) and exceptions (Section 6.2.5.2).

$$\frac{}{\theta \vdash c : c}\ (\textit{T-c}) \qquad \frac{\theta \vdash e_1 : t_1 \quad \theta \vdash e_2 : t_2}{\theta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \textbf{let}\ x = t_1\ \textbf{in}\ t_2}\ (\textit{T-let})$$

$$\frac{}{\theta \vdash x : x}\ (\textit{T-var}) \qquad \frac{\theta \vdash e_1 : t_1 \quad \theta \vdash e_2 : t_2 \quad \theta \vdash e_3 : t_3}{\theta \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3}\ (\textit{T-if})$$

$$\frac{\theta \vdash e_1 : t_1 \quad \theta \vdash e_2 : t_2}{\theta \vdash e_1\ e_2 : @_\theta(t_1, t_2)}\ (\textit{T-app}) \qquad \frac{\blacklozenge \vdash e : t \quad k\ \text{fresh}}{\theta \vdash \lambda x.e : \lambda_k x.t}\ (\textit{T-fun})$$

$$\frac{}{\theta \vdash \mathsf{print}\ e : \textbf{()}}\ (\textit{T-print}) \qquad \frac{}{\theta \vdash a : a}\ (\textit{T-addr})$$

$$\frac{i\ \text{fresh} \quad \theta \vdash e : t}{\theta \vdash \mathsf{ref}\ e : \textbf{ref}_i}\ (\textit{T-ref}) \qquad \frac{\rho\ \text{fresh} \quad \theta \vdash e : t}{\theta \vdash !e : \textbf{read}_\rho}\ (\textit{T-read})$$

$$\frac{}{\theta \vdash e_1{:=}e_2 : \textbf{()}}\ (\textit{T-a}) \qquad \frac{}{\theta \vdash \mathsf{raise} : \textbf{raise}_\theta}\ (\textit{T-raise})$$

$$\frac{\theta' \vdash e_1 : t_1 \quad \theta \vdash e_2 : t_2 \quad \theta'\ \text{fresh}}{\theta \vdash e_1\ \mathsf{handle}\ e_2 : t_1\ \textbf{handle}_{\theta'}\ t_2}\ (\textit{T-handle})$$

Figure 6.2: Term inference: constructing the term model for SiML.

$$\frac{}{\theta, E \vdash c : \bigcirc}\ (\textit{E-c}) \qquad \frac{\theta, E \vdash e_1 : \eta_1 \quad \theta, E \vdash e_2 : \eta_2}{\theta, E \vdash \textsf{let}\ x = e_1\ \textsf{in}\ e_2 : \eta_1 \triangleright \eta_2}\ (\textit{E-let})$$

$$\frac{}{\theta, E \vdash x : \bigcirc}\ (\textit{E-var}) \qquad \frac{\begin{array}{cc}\theta, E \vdash e_1 : \eta_1 & \theta, E \vdash e_3 : \eta_3 \\ \theta, E \vdash e_2 : \eta_2 & \theta \vdash e_1 : t_1\end{array}}{\theta, E \vdash \textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 : \eta_1 \triangleright \texttt{ite}(t_1, \eta_2, \eta_3)}\ (\textit{E-if})$$

$$\frac{}{\theta, E \vdash \lambda x.e : \bigcirc}\ (\textit{E-fun}) \qquad \frac{\begin{array}{cc}\theta, E \vdash e_1 : \eta_1 & \theta \vdash e_1\ e_2 : t \\ \theta, E \vdash e_2 : \eta_2 & E \vdash e_1\ e_2 :_{\mathrm{e}} \eta_{\mathrm{t}}\end{array}}{\theta, E \vdash \eta_1 \triangleright \eta_2 \triangleright (t : \eta_{\mathrm{t}})}\ (\textit{E-app})$$

$$\frac{\theta, E \vdash e : \eta \quad \theta \vdash e : t}{\theta, E \vdash \textsf{print}\ e : \eta \triangleright (t : \texttt{io})}\ (\textit{E-print}) \qquad \frac{\theta, E \vdash e : \eta \quad \theta \vdash \textsf{ref}\ e : \textbf{ref}_i\ t}{\theta, E \vdash \textsf{ref}\ e : \eta \triangleright (t : \textsf{a}_i)}\ (\textit{E-ref})$$

$$\frac{\begin{array}{c}\theta, E \vdash e : \eta \\ \theta \vdash e : t \\ \theta \vdash !e : \textbf{read}_\rho\end{array}}{\theta, E \vdash !e : \eta \triangleright (t : \texttt{r}_\rho)}\ (\textit{E-read}) \qquad \frac{\begin{array}{cc}\theta, E \vdash e_1 : \eta_1 & \theta \vdash e_1 : t_1 \\ \theta, E \vdash e_2 : \eta_2 & \theta \vdash e_2 : t_2\end{array}}{\theta, E \vdash e_1{:=}e_2 : \eta_1 \triangleright \eta_2 \triangleright (t_1 \texttt{:=} t_2 : \texttt{u})}\ (\textit{E-a})$$

$$\frac{\theta \vdash \textsf{raise} : \textsf{raise}_\theta}{\theta, E \vdash \textsf{raise} : \texttt{exn}_\theta}\ (\textit{E-raise})$$

$$\frac{\theta, E \vdash e_1 : \eta_1 \quad \theta, E \vdash e_2 : \eta_2}{\theta, E \vdash e_1\ \textsf{handle}\ e_2 : \eta_1 \triangleright \texttt{handle}_\theta\ \eta_2}\ (\textit{E-handle})$$

Figure 6.3: Precise effect analysis: constructing the effect model for SiML.

6.2.5.1 References and assignments

Reference values or 'ref-values' are updatable addresses, corresponding roughly to heap-allocated memory in imperative languages. SiML has three language constructs to support ref-values:

- Reference allocation, $\mathsf{ref}\ e$, which fully evaluates e to e', allocates a new address a, and maps a to e'. This operation has both a side effect (for allocating an address and mapping it to a value) and yields a computational result (the fresh address a).
- Assignment, $a{:=}e$, which evaluates a to a' and e to e' and then updates the store with the value e' at address a'. This operation is purely a side effect; its computational result is $\mathsf{()}$.
- Read, $!e$. This operation evaluates e to an address and extracts the value that this address points to. This operation has both a side effect and a computational result.

Thus, we have to capture all three in the effect model, and we also have to represent reference creation and reading in the term model. This is not entirely straightforward. Consider:

$$e_2 = \mathsf{let}\ x = !a\ \mathsf{in}\ (\mathsf{let}\ y = a := \mathsf{true}\ \mathsf{in}\ (\mathsf{let}\ z = !a\ \mathsf{in}\ x))$$

Here, we take an address a, read it out into x, then update a to point to true, read a out again into z, and then yield the old value of a written to x.

The effect model for e_2 would list a read, the assignment, and a read from the same address as the first read. Our term model, on the other hand, should give us the information that the computational result of this expression is identical to the result of reading a, in short: $e_2 : \mathbf{read}\ a$. But what if, in e_2, we return z instead of x? Clearly, the effect model is unchanged. But so is the term model: we are still returning the result of reading a — just at a different point in time!

We solve this problem by distinguishing between individual reads, annotating each with a unique *read index* ρ. As rule (*T-read*) in Figure 6.2 shows, we allocate a fresh ρ for every read; conversely, rule (*E-read*) in Figure 6.3 ensures that we use the same index at the appropriate location in the effect trace. For example, we might now represent the computational result of e_2 as $e_2 : \mathbf{read}_{\rho_1}$, but if we change e_2 to return z rather than x, it might become $\mathbf{read}_{\rho_2}$. The read index ρ_1 synchronises the term and effect model. Therefore, we need not repeat the address we are reading from (i.e., a) in the term model, since we can store and unambiguously look it up in the effect trace:

$$(a : \mathtt{r}_{\rho_1}) \triangleright (a{:=}\mathsf{true} : \mathsf{u}) \triangleright (a : \mathtt{r}_{\rho_2})$$

The first entry in our effect trace is $a : \mathtt{r}_{\rho_1}$, representing a read effect on precisely the address a.

Our trace above also shows the effect trace representation for assignments, as $a{:=}\mathsf{true} : \mathsf{u}$. Here, u is the effect type of assignments, with the actual assignment, $a{:=}\mathsf{true}$, indicating the term that we are assigning and the term we are assigning to.

Reference allocation (via ref) has precisely the same issues as read: whenever we allocate references for the same value, we cannot distinguish them in the term model alone. We use the same solution as for read, namely tagging each allocation term and allocation effect by a unique *read index* i. For example, we might represent an expression $\mathsf{ref}\ \mathsf{true}$ as the term $\mathbf{ref}_i$ with an effect trace of $(\mathsf{true} : \mathsf{a}_i)$, where a is the effect type for allocation.

6.2.5.2 Exceptions

Exceptions affect both the computational result of a term and its side effect behaviour. Consider the program

$$\mathsf{let}\ x = \mathsf{raise}\ \mathsf{in}\ A$$

Here, our exception skips all effects and computational results in A. Conversely, in

$$\mathsf{true}\ \mathsf{handle}(\mathsf{let}\ x = \mathsf{print}\ \mathsf{true}\ \mathsf{in}\ \mathsf{false})$$

the *lack* of an exception means that we will skip both the effect and the computational effect in the exception handling branch. As we see, exceptions and their handlers both affect term and effect model. Since we do not distinguish between different kinds of exceptions in SiML, it is not necessary to uniquely associate exceptions in the term model with exceptions in the effect trace.

The same does not hold for exception handlers. Consider the following programs:

$$\begin{aligned} e_3 &= (\mathsf{let}\ x = \mathsf{raise}\ \mathsf{in}\ \mathsf{raise})\ \mathsf{handle}\ \mathsf{true} \\ e_4 &= \mathsf{let}\ x = \mathsf{raise}\ \mathsf{in}\ \mathsf{raise}\ \mathsf{handle}\ \mathsf{true} \end{aligned}$$

Here, e_3 raises an exception within an exception handler and thereby yields true. However, e_4 raises an exception *before* entering an exception handler, and thus terminates exceptionally.

Both e_3 and e_4 have the term model **raise handle** true, representing their computational result. However, if we put all of their effects into order, what we get for both is

$$(\textbf{raise} : \mathtt{exn}) \triangleright (\textbf{raise} : \mathtt{exn}) \triangleright (\mathtt{handle}\ \circ)$$

i.e., two exceptions (exn here is the type for exceptions), followed by an exception handler, a special language construct $\mathtt{handle}\ \eta$, where η captures all the effects in the exception handler's exceptional branch. In our examples, the exception handler has no effects if it triggers; we mark this as $\circ$, representing 'no effect.'

Clearly, the straightforward solution is insufficient: e_3 and e_4 compute different things, so they should differ in at least one of the term model or the effect model.

The difference between e_3 and e_4 is, of course, that in e_3 all exceptions are syntactically surrounded by the exception handler, whereas in e_4 the first exception 'escapes,' giving two very different control flows. The exceptions thus have different meanings: one has the meaning of 'jump to the surrounding exception handler', and one has the meaning of 'continue, as exception, to whatever called me.' To distinguish these different meanings, we make them manifest with an annotation, a *handler ID* θ. This θ matches each exception to its (similarly annotated) matching exception handler. If there is no surrounding exception handler, we use the special handler ID $\blacklozenge$. In our example, we now get

$$(\textbf{raise} : \mathtt{exn}_\theta) \triangleright (\textbf{raise} : \mathtt{exn}_\theta) \triangleright (\mathtt{handle}_\theta \circ)$$

for e_3, whereas e_4 would have the effect trace

$$(\textbf{raise} : \mathtt{exn}_\blacklozenge) \triangleright (\textbf{raise} : \mathtt{exn}_\theta) \triangleright (\mathtt{handle}_\theta \circ)$$

making clear that the initial exception will escape the handler. This now explains our use of θ to the left-hand side of our turnstile in term and effect inference (Figures 6.2 and 6.3): we begin with $\theta = \blacklozenge$, but for each exception handler we introduce a fresh handler ID θ' in (*T-handle*) and re-use it in (*E-handle*).

Note that raise is not the only possible source for an exception: function applications might also raise an exception. Thus, we make sure that our rule (*T-app*) annotates each function application @ with the surrounding handler ID, too.

6.2.5.3 Functions

Functions are challenging for another reason: they can be recursive. How, then, can we hope to compute a finite effect trace for a function application? The best we can do is to conservatively approximate the effects of a function. In rule (*E-app*) (Figure 6.3), the effect trace we give to function applications is therefore the function application term *itself* with an approximated exception type. We obtain this approximated type in rule (*E-app*), as η_t through the judgement $E \vdash e_1\ e_2 :_e \eta_t$.

This judgement relies on the analysis in Figures 6.4 and Figure 6.5, which is a straightforward effect analysis. In Figure 6.4, we describe rules $E \vdash e :_e \eta$ that conservatively approximate the effects of expression e as η, with E acting as an environment, whilst in Figure 6.5 we support this analysis by inferring effect signatures $E \vdash e : \tau_e$ that describe the *potential* effect of a function value, triggered when we apply the function to a parameter.

This inference can yield only one of two possible types for η_t: $\circ$ (meaning 'no effect, the function is pure') or $\star$ (meaning 'we might encounter any effect here'). Together with our other effect types, these two form the following type lattice:

$$\begin{tikzcd}
 & & \circ \arrow[dll, dash] \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
\mathtt{a}_i & \mathtt{u} & \mathtt{r}_\rho & \mathtt{io} & \mathtt{exn}_\theta \\
 & & \star \arrow[ull, dash] \arrow[ul, dash] \arrow[u, dash] \arrow[ur, dash] \arrow[urr, dash] & &
\end{tikzcd}$$

The lattice we use in our implementation is more detailed and sound, cf. Section 8.3.1.1, as well as parametric [TJ91], though the above is sufficient for our purposes here.

Our definitions in Figures 6.4 and 6.5 guarantee that the type inference-based judgements $E \vdash e :_e \eta$ conservatively approximate our precise effect analysis (Figure 6.3); this is an important property in our soundness proof (Appendix D).

Since we treat *all* function applications conservatively, we would have very imprecise results for expressions such as

$$(\lambda y.A)\ \mathbf{()}$$

where all interesting things happen in A, if we were not to compute an effect trace for A. This imprecision affects all function abstractions, so we remedy it by annotating each function abstraction with its effect trace. In our term model, we annotate each lambda with a unique *lambda key* k, which we then map to the effect trace of that function's body.

Figure 6.6 summarises the syntax for our term and effect language. The only construct listed there that we did not discuss is the `ite` construct, the conditional on effect traces: this conditional is analogous to the conditional on terms.

$$\frac{}{E \vdash c :_{\mathrm{e}} \bigcirc} \qquad \frac{\begin{array}{ll} E \vdash e_1 :_{\mathrm{e}} \varepsilon_1 & \varepsilon' <:_{\mathrm{e}} \varepsilon_1 \\ E \vdash e_2 :_{\mathrm{e}} \varepsilon_2 & \varepsilon' <:_{\mathrm{e}} \varepsilon_2 \\ E \vdash e_3 :_{\mathrm{e}} \varepsilon_3 & \varepsilon' <:_{\mathrm{e}} \varepsilon_3 \end{array}}{E \vdash \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 :_{\mathrm{e}} \varepsilon'}$$

$$\frac{}{E \vdash x :_{\mathrm{e}} \bigcirc}$$

$$\frac{}{E \vdash \lambda x.e :_{\mathrm{e}} \bigcirc} \qquad \frac{\begin{array}{ll} E \vdash e_1 : \alpha \overset{\varepsilon}{\rightarrow} \beta & \varepsilon' <:_{\mathrm{e}} \varepsilon \\ E \vdash e_1 :_{\mathrm{e}} \varepsilon_1 & \varepsilon' <:_{\mathrm{e}} \varepsilon_1 \\ E \vdash e_2 :_{\mathrm{e}} \varepsilon_2 & \varepsilon' <:_{\mathrm{e}} \varepsilon_2 \end{array}}{E \vdash e_1\ e_2 :_{\mathrm{e}} \varepsilon'}$$

$$\frac{}{E \vdash \textsf{print } e :_{\mathrm{e}} \star}$$

$$\frac{}{E \vdash \textsf{ref } e :_{\mathrm{e}} \star} \qquad \frac{}{E \vdash !e :_{\mathrm{e}} \star} \qquad \frac{}{E \vdash e_1 := e_2 :_{\mathrm{e}} \star}$$

$$\frac{}{E \vdash \textsf{raise} :_{\mathrm{e}} \star} \qquad \frac{\begin{array}{ll} E \vdash e_1 :_{\mathrm{e}} \varepsilon_1 & \varepsilon' <:_{\mathrm{e}} \varepsilon_1 \\ E \vdash e_2 :_{\mathrm{e}} \varepsilon_2 & \varepsilon' <:_{\mathrm{e}} \varepsilon_2 \end{array}}{E \vdash e_1 \textsf{ handle } e_2 :_{\mathrm{e}} \varepsilon'}$$

$$\frac{E \vdash e_1 :_{\mathrm{e}} \varepsilon_1 \quad E \vdash e_2 :_{\mathrm{e}} \varepsilon_2 \quad \varepsilon' <:_{\mathrm{e}} \varepsilon_1 \quad \varepsilon' <:_{\mathrm{e}} \varepsilon_2}{E \vdash \textsf{let } x = e_1 \textsf{ in } e_2 :_{\mathrm{e}} \varepsilon'}$$

Figure 6.4: Abstract effect inference: approximating the effect model for SiML. Here, ($<:_{\mathrm{e}}$) introduces subtype constraints.

$$\frac{}{E \vdash c : \bigcirc} \qquad \frac{}{E \vdash x : E(x)} \qquad \frac{E\{x \mapsto \varepsilon\} \vdash e :_{\mathrm{e}} \varepsilon}{E \vdash \lambda x.e : \alpha \overset{\varepsilon}{\rightarrow} \beta}$$

$$\frac{}{E \vdash \textsf{print } e : \star} \qquad \frac{E \vdash e_1 : \alpha \overset{\varepsilon}{\rightarrow} \beta \quad E \vdash e_2 <: \alpha}{E \vdash e_1\ e_2 : \beta}$$

$$\frac{E \vdash e_2 : \alpha \quad E \vdash e_3 : \beta \quad \gamma <: \alpha \quad \gamma <: \beta}{E \vdash \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 : \gamma}$$

$$\frac{}{E \vdash \textsf{ref } e : \star} \qquad \frac{}{E \vdash !e : \star} \qquad \frac{}{E \vdash e_1 := e_2 : \star}$$

$$\frac{}{E \vdash \textsf{raise} : \star} \qquad \frac{\begin{array}{ll} E \vdash e_1 : \alpha & \gamma <: \alpha \\ E \vdash e_2 : \beta & \gamma <: \beta \end{array}}{E \vdash e_1 \textsf{ handle } e_2 : \gamma}$$

$$\frac{E \vdash e_1 : \alpha \quad E \vdash e_2 : \beta \quad \gamma <: \alpha \quad \gamma <: \beta}{E \vdash \textsf{let } x = e_1 \textsf{ in } e_2 : \gamma}$$

Figure 6.5: Abstract effect inference: effect signatures.

$$\begin{array}{lll}
t \in \mathit{Term} & ::= c \mid x \mid a \mid @_\theta(t_1, t_2) \mid \lambda_k x.t \mid \textbf{ref}_i \mid \textbf{read}_\rho & \\
 & \quad \mid \textbf{raise}_\theta \mid t_1\ \textbf{handle}_\theta\ t_2 \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 & \\
t_\mathrm{e} & ::= t_1 \mathtt{:=} t_2 \mid t & \\
\eta \in \mathit{Effect} & ::= t_\mathrm{e} : \tau_\mathrm{e} \mid \bigcirc \mid \mathtt{ite}(t, e_1, e_2) & \\
 & \quad \mid \mathtt{handle}_\theta\ e_2 \mid \eta_1 \triangleright \eta_2 & \\
\tau_\mathrm{e} \in \mathit{EffectTy} & ::= \bigcirc \mid \star \mid \mathtt{a}_i \mid \mathtt{u} \mid \mathtt{r}_\rho \mid \mathtt{io} \mid \mathtt{exn}_\theta & \\
\theta \in \{\blacklozenge, \ldots\} & & \text{(Exception handler ID)} \\
k & & \text{(Lambda Key)} \\
i & & \text{(Address ID)} \\
\rho & & \text{(Read ID)}
\end{array}$$

Figure 6.6: Term and effect languages.

6.2.5.4 Equivalences among terms and effects

So far, we have described how our program models ensure that we do not treat two distinct programs as equal. To algorithmically optimise programs, we must relax these checks somewhat: ideally we would like to allow the user to replace any subexpression by an equivalent subexpression without our models detecting false inconsistencies.

We therefore relax model equivalence in three ways. First, we allow alpha conversion for all names, lambda keys, handler IDs, and read/allocation indices. Second, we permit user-specified equivalences, as we detailed in Section 5.2. Third, we allow a selection of term and effect equivalences that intrinsically preserve behaviour due to the programming language semantics.

Figure 6.7 lists some of the equivalences that apply to SiML. The first four equivalences affect the term model. (TE-if-t) and (TE-if-f) encode the semantics of the conditional, allowing the user to replace, for example, a term (**if** true **then** x **else** y) by y, while the next two rules handle beta reduction and let term substitution, respectively. Again, recall that our effect model ensures, in separation, that these equivalences do not alter the side-effect behaviour of our program.

This can prohibit us from some behaviour-preserving changes. Consider the following program with its term model:

$$(\lambda x.x)\mathsf{true} : @_\blacklozenge(\lambda_k x.x, \mathsf{true})$$

This program has the effect trace $@_\blacklozenge(\lambda_k x.x, \mathsf{true}) : \bigcirc$. If we now try to transform this program to true through rule (TE-β), our effect trace will be empty!

Of course, what $@_\theta(\ldots) : \bigcirc$ really means is that we don't have any effect — just as with an empty effect trace (which we denote $\bigcirc$, making this symbol do double duty). We address this purely syntactic discrepancy by introducing a number of 'clean-up' equivalence rules that affect our effect traces. First, rule (EE-app-$\bigcirc$) eliminates a pure application from the effect trace and thereby permits our pure beta reduction example from above. The next three rules we list in Figure 6.7 are cleanup rules that allow us to eliminate such non-effects $\bigcirc$ if they occur in the middle of an effect trace.

Rules (EE-if-t) and (EE-if-f) are the analogues to our conditional support for terms. The remaining rules are more interesting:

$$
\begin{array}{llcll}
(a) & \textbf{if}\ \text{true}\ \textbf{then}\ t_2\ \textbf{else}\ t_3 & = & t_2 & \text{(TE-if-t)} \\
 & \textbf{if}\ \text{false}\ \textbf{then}\ t_2\ \textbf{else}\ t_3 & = & t_3 & \text{(TE-if-f)} \\
 & @_\theta(\lambda_k x.t_1, t_2) & = & t_1[x\backslash t_2] & \text{(TE-}\beta\text{)} \\
 & \textbf{let}\ x = t_1\ \textbf{in}\ t_2 & = & t_2[x\backslash t_1] & \text{(TE-let)}
\end{array}
$$

$$
\begin{array}{lcl}
(b) & t_1\ t_2 : \bigcirc = \bigcirc & \text{(EE-app-}\bigcirc\text{)} \\
 & \bigcirc \triangleright \eta = \eta & \text{(EE-}\ \bigcirc\ \text{-l)} \\
 & \eta \triangleright \bigcirc = \eta & \text{(EE-}\ \bigcirc\ \text{-r)} \\
 & \mathtt{ite}(t, \bigcirc, \bigcirc) = \bigcirc & \text{(EE-if-}\bigcirc\text{)} \\
 & \mathtt{ite}(\text{true}, \eta_t, \eta_f) = \eta_t & \text{(EE-if-t)} \\
 & \mathtt{ite}(\text{false}, \eta_t, \eta_f) = \eta_f & \text{(EE-if-f)} \\
 & (t_1 : \mathtt{r}_i) \triangleright (t_2 : \mathtt{r}_j) = (t_2 : \mathtt{r}_j) \triangleright (t_1 : \mathtt{r}_i) & \text{(EE-read)} \\
 & (t : \mathsf{a}_i) \triangleright x = x \triangleright (t : \mathsf{a}_i) & \text{(EE-ref)} \\
 & (t_1 \mathtt{:=} t_1') \triangleright (t_2 : \mathtt{io}) = (t_2 : \mathtt{io}) \triangleright (t_1 \mathtt{:=} t_2) & \text{(EE-ioa)}
\end{array}
$$

Figure 6.7: Term (a) and effect (b) equivalence rules.

Rule (EE-read) allows us to re-order reads. Since reads do not change anything, this re-ordering is safe. Rule (EE-ref) similarly allows us to re-order allocations with anything. Since allocations generate a *fresh* entry on the heap, their actions do not conflict with any other actions. Finally, (EE-ioa) allows us to re-order prints and assignments, which affect separate parts of the state. Note that this is perfectly safe even if we print the result of reading a value: reading is an effect, too, and we may not re-order reads and assignments.

6.2.5.5 Soundness

As we prove in Appendix D, our combined models guarantee full behaviour preservation, with the following exceptions:

- We do *not* guarantee that the program's termination behaviour will be preserved, and
- We do *not* validate that any user-specified axioms are correct.

The second point is an intrinsic limitation in any system that accepts user-specified axioms. Note that there are existing approaches towards validating various kinds of specifications, including automatic test generation and checking [DF94, CTCC98], concurrent program execution and specification interpretation [HRD08], and proof of correctness through a proof assistant such as Isabelle or Coq. We consider such validation to be outside the scope of this work.

6.2.6 Congruence polarity model

The congruence polarity model implements our congruence casts and polarities from Section 5.2.5. This model attempts to understand term mismatches by introducing congruence casts, and where it succeeds in doing so, it assigns polarity types that are a straightforward extension of the types we described in Section 5.3 to the full SML type system.

We maintain the congruence casts as part of our term model and issue two forms of inconsistency: congruence mismatches (corresponding to type errors on polarity types), and public polarity changes that we can best express as part of the external interface metamodel, the final program model we have used in our prototypes.

6.2.7 External interface metamodel

In the description of our models we have so far pretended that we must preserve behaviour for all attributes on each AST node, intermediate model node, or label, but in practice there is no need to preserve the behaviour of definitions that are neither externally visible nor directly affect externally visible attributes.

Conversely, all such externally visible properties must be subjected to additional rigour: they must preserve their names as well as other relevant properties, such as their types (for typed languages), visibility (for languages with language support for data hiding), value type (for SML), type class bindings (for Haskell) and so on.

These observations separate the attributes in our program models into two classes: *externally visible* attributes and *externally invisible* attributes. Here, the externally invisible attributes of a program are precisely the attributes that our users can change without altering the program's behaviour.

For example, consider the following SML module:

```
fun sort (data) =
    let fun compare (x, y) = x > y
        fun dump (x) = print (Data.toString (x))
    in  SortLib.merge_sort (compare) (data)
    end
```

Here, sort is externally visible: any external program can rely on this module exporting the function sort, and on that function having a certain type[2] . Hence the name sort, the definition of this function, and the type (and other assorted properties) are all externally visible attributes. In the body of sort, we define two helper functions, compare and dump. Both functions are hidden inside the body of sort, so their names and types are not externally visible. Here, compare contributes to the computational result of sort; consequently, all attributes in compare that contribute to the computational result are visible. Function dump, presumably a leftover from an earlier implementation, is made up of *dead attributes* since it does not contribute to the program behaviour in any way.

The situation is less straightforward in e.g. Java:

```
public class C
{
      private int i;
      private int obsolete;
      protected C(int j) { i = j; }
      public int get() { return i; }
}
```

Clearly, get is in the public interface, and so its attributes are visible to anyone. Similarly, the constructor C(int) is visible to all subclasses and thus interface code. But so is i, despite being private, since we can write a program that detects whether its name has changed (using reflection). For the same reason, the field obsolete — despite being unused — is entirely visible.

From the perspective of program metamorphosis, the classification into these two categories affects how we report differences: we can ignore changes on invisible attributes, but must (or, at least, should) report changes on visible ones.

In our SML system, we therefore expressly distinguish between externally visible and externally invisible change and report term model changes only if they affect an externally visible entity; as we mentioned, we complement this mechanism by also reporting congruence polarity changes in externally visible definitions. However, we make no attempt to preserve names in the same fashion.

For some properties, it may be easier to maintain a separate model to enforce the external interface (Section 6.3.1).

6.3 Other feasible program models

We now discuss models that we have conceptually explored, but not used in any of our prototypes. We expect that many more of the program analyses that we traditionally use for identifying optimisation opportunities or programming errors can be adapted into program models.

[2] The external program can also rely on sort not being a value constructor or exception, which can be relevant during pattern matching.

6.3.1 Explicit API model

The explicit API model is an example of a label association model. This model would capture the external API of all modules, and ensure that this API is preserved — irrespectively of whether the names are bound to the same functionality before and after transformation.

We considered this model for our Java prototype but decided against it, both because Java semantics would require us to yield an inconsistency after almost all renamings, and because we found that a judicious use of the external interface metamodel (Section 6.2.7) would be more precise if we wanted to add this class of behavioural guarantees.

6.3.2 Points-to model

The points-to model would identify whether the set of values that a given reference might point to has changed. We expect that such a model might be useful to aid programmers in understanding inconsistencies in heavily imperative code when the inconsistencies that our effect model (Section 6.2.3) would issue would contain too many false positives or be too hard to grasp for other reasons. In our experiments, we did not observe the need for such a model due to our choice of benchmarks, so we did not explore this option further.

6.3.3 Constrained-type name models

Constrained-type name models are a refinement over the name models we described earlier. Their purpose is to increase precision for object-oriented languages, particularly in the presence of delegation.

Recall that for an expression such as s.f, a simple name model will use the static type of s to determine the meaning of f. With a constrained-type name model, we would instead use a points-to or flow analysis to determine all possible *dynamic* types for s and then compare the set of possible meanings of f between the current and desired program models.

6.3.4 Parametric name models

A parametric name model is another possible approach to name model refinement. Consider again the case of s.f. With a parametric name model, we use the dynamic type of s whenever s is constant, and default (typically to the static type) otherwise.

Consider the following program:

```
class A { int g () { ... } }
class B extends A { int g () { ... } }
class C {
    final int h(A a)
    {
        return a.g(); // Site (1)
    }

    int main()
    {
        return h(new B()); // Site (2)
    }
}
```

In this program, main calls h at site (2) with an object of dynamic type B and static type A. At site (1) we only see the static type. However, at site (1) we can also see that our caller may know the dynamic type, and might be able to give us more precise type information.

Here, this assumption indeed holds: site (2) is instantiating a B, so the dynamic type must be B.

The idea behind a parametric name model is to exploit this insight in the style of procedure summaries [YYC08]. For any method h we identify *constant parameters*, i.e., parameters that are never assigned to. Any dispatch through a constant parameter we check not in h itself, but at all of h's call sites (where these checks may again be deferred to the calling function's call sites.)

This idea of incorporating context by deferring identifier matches to the call site extends to entire objects: any 'final' field that is assigned a constant parameter from the perspective of all constructors behaves just like a constant parameter for a method, except by affecting all dispatches through that field within the class (but not in its subclasses, since those have their own constructors).

In practice, this approach faces a number of challenges, since we cannot always determine the callee and may have to make whole-program assumptions to combine the information about possible callees.

We leave the details of parametric name models to future work.

6.4 Model patterns

Amongst our program models, we observed several ideas and design principles that were either common between models or lent themselves well to generalisation; we summarise them below.

6.4.1 Cartesian product model

Our Java prototype used two independent program models, specifically a name model and a Use/Def model. This illustrates the most straightforward way to build one program model out of several, namely to combine the component models into a *cartesian product*, such that the models are not aware of or interact with each other.

6.4.2 Intermediate program model

In Section 6.1.1 we suggested that some models can benefit from common program analysis results. For example, in our SML prototype we normalise the object program into a simpler representation that eliminates certain intermediate constructions (such as if or case expressions, whose semantics match those of function applications with pattern matching) and enrich this representation name analysis and type inference results. We identify each component of this intermediate program model with an AST node (to simplify inconsistency reporting); thus, our term and effect models (Section 6.2.3) and our congruence polarity model (Section 6.2.6), which are based on this intermediate model, have easy access to AST nodes.

Note that intermediate program models often must retain 'obsolete' program fragments. As we mentioned in Section 6.1.2, inconsistencies may sometimes depend on obsolete program components; in our SML prototype, we had to retain obsolete term and effect information in order to produce useful inconsistency reports.

6.4.3 (Recursive) behavioural map model

Several models with k-association (AST association, label association etc.) have the structure of a finite map from *k* to terms that may contain further *k*. Examples of this structure are the term model (Section 6.2.3) and our Java prototype's Use/Def model (Section 6.2.2).

Without formally defining them in detail here, we note that recursive map models have a useful property that simplifies change localisation (Section 6.1.2), since they admit attribute isolation as a means for avoiding cascading change.

6.4.4 Model refinement

Some models can make other models more precise. For example, in our SML prototype, our term model increases the precision of our name model. Consider the program below:

```
1 fun f (x) = x + 1
2 fun g (y) = y + 1
3 val h = f (1)
```

If we rename g in line 2 to f, we introduce a name capture inconsistency in the name model in line 3. However, our term model can show (through inlining) that the program before and after this transformation still computes the same result in h, since the functions f and g compute the same result.

We may still choose to issue an inconsistency here, if we heuristically assume that name capture is usually accidental. If the programmer had wanted to replace the call to f by a call to g in line 3, why did they not replace f by g in line 3 instead of the rather less transparent renaming? Indeed the fact that we have two identical functions here suggests that the user might be about to delete one or change its behaviour. If we silently accept the name capture and the user then changes the body of the function in line 2, we might now report an inconsistency in line 3 — even though the programmer never touched line 3, nor any of the functions originally called in this line!

Our SML prototype takes the above view. Specifically, if our user introduces a name capture as in the above, we will issue a name capture inconsistency. Only after the user accepts this behavioural change will we consult the term model to see whether we should now issue a term inconsistency; until then, our system will internally retain the identifier for f in line 3.

The advantage of gradual refinement is that it allows us to heuristically flag likely 'accidents' without requiring a complex (and, possibly, slower) analysis to determine whether our user has really changed the program behaviour. The disadvantage of this approach is that if we pick poor heuristics, we may issue many false positives and thus needlessly slow down the metamorphosis process.

Note that the congruence polarity model (Section 6.2.6) is another instance of model refinement; we return to it in the next section.

6.5 Interacting with the program model

For refactoring and related tasks, the program model serves only to highlight behavioural change. However, once users begin to evolve their program's behaviour, we must permit them to update the model.

We have observed three usage scenarios through which a user would manipulate the program model: accepting change (Section 6.5.1), choosing a possible explanation during an inconsistency (Section 6.5.2), and explicitly setting a goal model (Section 6.5.3). We have only implemented the first, accepting change, though in the following we illustrate how the other interaction modes may be of use in a production-quality program metamorphosis system.

6.5.1 Accept change

Whenever there is a discrepancy between a current model property and a desired model property, accepting change will copy the current property into the desired property. Most of the individual program models we have observed follow the (recursive) behavioural map model, which means that this will fully isolate the change.

However, for some models, such as the congruence polarity model's external interface model, this is not so easy. In that case, we report external polarity change inconsistencies for all externally visible definitions that our type inference deems not to be 'neutral' (e.g. because the programmer changed them from a list to a vector.) Since neutrality is the result of an inference mechanism, we cannot easily isolate this change. Instead we record a set of AST nodes for which we suppress this particular inconsistency. Once all such inconsistencies are suppressed AND the congruence polarity model type-checks, we can safely eliminate all congruence casts and clear the set of AST nodes for which we suppress polarity changes.

6.5.2 Choose explanation

In Chapter 5 we explained how our system uses congruence casts to locally explain transitions between types or algorithms. We further mentioned that our SML prototype heuristically chooses a congruence-based explanation, if there are multiple options. In practice, this heuristic choice may be incorrect. It is even possible that we might be able to explain a change as a congruence when the user merely intended a replacement with no deeper agenda.

We anticipate that a full-fledged program metamorphosis system will give users the option to choose between different explanations for congruence casts, or even to tell the system a priori that they intend e.g. a 'list to set' transition.

6.5.3 Set goal model

Finally, a program metamorphosis system may allow users to explicitly specify a goal model for their transformation that is not directly derived from the program. For example, a user might tell the program model that she is defining a function that must meet a certain specification, such as having no side effects. In this scenario, the user would employ our equivalence checks proactively, rather than as a safety net.

This step bridges the gap between program metamorphosis and program refinement.

Chapter 7

Building program metamorphosis tools

Building a program metamorphosis system is similar in many ways to building a compiler or a refactoring engine. Specifically, we must provide a *program analysis infrastructure*:

- a parser,
- an abstract program representation suitable for program analysis (typically an abstract syntax tree or AST), and
- program analyses that reveal information about the program behaviour.

Unlike a compiler, a program metamorphosis system needs no backend. Instead, it requires an *interactive program transformation infrastructure*, something it shares with refactoring engines:

- a user interface,
- a concrete program representation that captures the full source program (including comments and whitespace), typically a concrete syntax tree or CST with whitespace annotations, and
- syntactic transformations on the concrete syntax tree.

A refactoring engine is now a program analysis infrastructure together with an interactive program transformation infrastructure, plus implementations of the refactoring precondition and accompanying transformations (e.g., using an extended version of the techniques suggested in GenTL [AK08]).

A program metamorphosis system has almost the same needs: it requires a program analysis infrastructure and an interactive program transformation infrastructure, but unlike preconditions tupled with transformations, the program metamorphosis system requires

- a program model (which can be a combination of program models, as we described in the previous chapter) and
- program model updates for each transformation.

While it is certainly possible to implement all of these components by hand (as two of our prototypes do), it may be possible to provide improved support for the needs of program metamorphosis by defining a framework for developing program metamorphosis systems. We have taken initial steps in this direction, both as a basis for program metamorphosis implementation and as a basis for future research into program metamorphosis, as we describe towards the end of this dissertation in Sections 10.1, 10.3, and 10.4.

Our infrastructure support for program metamorphosis currently comprises the components for an attribute tree grammar program analysis system [Knu68]. As with existing compiler construction tools (Section 4.1.3), we support all components needed for a program analysis infrastructure, but require program analyses to be expressed in terms of attribute tree grammars.

Attribute computations in our language take the form of computations on six kinds of values:

- *terminals*, such as variable names, typed by the regular expression that describes their contents,
- *AST nodes*, typed by the set of grammar productions that may produce them,
- *strings*,
- *fresh values*, typed uniquely,
- *user-defined algebraic values*, defined by (possibly recursive) user-defined algebraic datatypes, and
- *built-in datatypes*, specifically booleans, lists, sets, and maps.

We note that with the exception of the string formatting operator built into our language, all AST computations could in theory be phrased in terms of Datalog. This choice is deliberate and may facilitate *recovery planning*, one of the possible avenues for future work (Section 10.3).

During our implementation, we found that attribute grammars offer advantages to the program metamorphosis approach, but are also too limited for some of the tasks a program metamorphosis system must handle. Below we list the advantages and disadvantages we observed.

Advantages. Where attribute tree grammars are expressive enough for our needs, they do simplify the specification of program metamorphosis systems:

- *Easy desired program models*: once we know what attributes we must preserve, we can simply copy them into a 'desired' attribute for the current tree node. If this desired attribute is set but the current one disagrees, we have identified a behavioural change. Accepting this change then simply means that we copy the local current attribute into the desired attribute.

  Furthermore, if the user copies or moves an AST fragment, we can easily retain the attributes, since they are tied to the concrete AST nodes.

- *Change localisation*: all computations are attribute computations on a given tree node, we can always attribute inconsistencies to whichever computation no longer reproduced the desired result for any given attribute.

Disadvantages. Attribute tree grammars are not expressive enough for some tasks that we encounter when specifying real-life programming languages. Furthermore, the advantages they offer may be too limited for practical application.

- *Limits to change localisation*: the implicit change localisation in attribute tree grammars is sometimes too limited, particularly for issuing more complex inconsistencies (such as name capture inconsistencies). We normally detect these at the location at which the capture occurs, but to understand such an inconsistency users should also see both of the definitions involved in the capture — the intended definition, and the actual definition.

- *Limits on expressivity*: standard computations for attribute grammars do not include fixed point computations or similar, more involved program analyses. Thus, we cannot (for example) implement ML-style type inference directly.

- *Limits on tree structure*: some languages, such as Haskell and SML, allow user-defined operator fixities, meaning that the structure of the AST depends, in part, on the results of semantic analysis. Thus, we cannot separate parsing and semantic analysis as cleanly as attribute tree computations normally expect.

Our system provides a number of plugins that extend basic attribute grammar computations, including plugins for type inference and fixity parsing. Our Standard ML prototype (Section 8.3) uses the latter plugin extensively. Earlier versions of our prototype also used the type inference plugin, though we eliminated this use when we changed the bulk of our program analyses to operate on an intermediate program model (Section 6.4.2).

Appendix F illustrates how we can specify a small programming language, $L0$, with our system. This illustrates both our syntactic mechanisms and some of our semantic analyses, including the type inference plugin.

Appendix G, the documentation for our tool, describes its facilities in more detail.

Chapter 8

Prototypes

"Listen, lad. I built this kingdom up from nothing. When I started here, all there was was swamp. Other kings said I was daft to build a castle on a swamp, but I built it all the same, just to show 'em. It sank into the swamp. So, I built a second one. That sank into the swamp. So I built a third one. That burned down, fell over, then sank into the swamp."
— The Father (Michael Palin), Monty Python and the Holy Grail

We now describe our three prototype implementations in chronological order. We begin with our initial prototype, which we refer to as the *Datalog* prototype, in Section 8.1. This prototype handles nominal and some simple structural transformations on Standard ML minus the module language and maintains a single global program model, represented as a Datalog fact database.

The second prototype is a Java prototype, developed as a plugin for the Eclipse platform. This prototype uses a completely manual implementation of the program model. We describe it in Section 8.2.

Our third prototype again targets Standard ML, with almost complete support for the entire language and a more flexible set of transformations than in our initial prototype. This prototype uses the attribute grammar mechanism we discussed in the previous section for part of its program model, and manual coding for the rest; we describe it Section 8.3.

For each prototype, we describe the choice and implementation of our program models, the frontend we used, and the set of transformations we support.

8.1 Datalog prototype

Our initial prototype handles the Standard ML core language (without modules). We implemented it on top of the SML of New Jersey (SML/NJ) frontend, modifying the ml-yacc parser generator to retain a concrete syntax tree of each processed program for later manipulation.

After program analysis, our system emits a number of Datalog ground facts (EDB facts, in Datalog terminology) that capture the naming and scoping model; we then used a Datalog logic program in an attempt to infer inconsistencies.

In the following, we give a full description of a version of our system that we restricted to *Scope ML*, a subset of SML that is restricted to questions of scoping (Appendix A), i.e., value definitions and uses.

Relation	Meaning
$\ell \prec \ell'$	Location ℓ immediately precedes ℓ' in scope
$i \overset{\mathsf{n}}{\mapsto} n$	Identifier i has name n
$\mathsf{Use}(i, \ell)$	Identifier i used at location ℓ
$\mathsf{Def}(i, \ell)$	The definition of identifier i is at location ℓ
$\mathsf{PDef}(i, \ell)$	Identifier i defined at ℓ as a parameter
$\ell_h \triangleleft \ell_b$	ℓ_b is definition branch of definition head ℓ_h
$\mathsf{Head}(\ell)$	Location ℓ is a definition head
$\mathsf{ValHead}(\ell)$	Location ℓ is a **val** definition head
$\mathsf{FunHead}(\ell)$	Location ℓ is a **fun** definition head
$\mathsf{Val}(\ell)$	Location ℓ is a **val** definition branch
$\mathsf{Fun}(\ell)$	Location ℓ is a **fun** definition branch

Table 8.1: Ground relations for Scope ML.

8.1.1 Program model

For this prototype, we define the full logical structure of the name model through relations, to give a fully detailed example of how such a model works and what kind of information it requires from the programming language frontend. We build our relations between the following sorts: the sort of names, whose instances we give as n, the sort of identifiers, whose instances we give as i, and the sort of locations, expressed as ℓ.

Table 8.1 describes a family of relations that we extract from the SML/NJ name analysis. A name model (in this context) is precisely such a family. Note that we make locations and scope very explicit in the above: for example, our relation $\ell \prec \ell'$ captures the structure of the referencing environment, stating that location ℓ' can see all definitions from location ℓ. The relation ℓ thereby constructs a graph that represents the scoping structure in our program; this graph is acyclic if we only have nonrecursive definitions but may contain cycles if definitions can see themselves.

Some of the remaining definitions are straightforward, others require elaboration:

- $\mathsf{Def}(i, \ell)$ and $\ell \triangleleft \ell'$: the relations ($\triangleleft$) and Def together represent the definitions of identifiers with nontrivial scoping rules. Definitions that we make via $\mathsf{Def}(-, \ell)$ and $\ell_h \triangleleft \ell$ are visible to all locations that can reach location ℓ_h via ($\prec$). This mechanism allows us to represent recursive and non-recursive definitions. Consider the following example:

$$\begin{array}{lll} \textcircled{\scriptsize ℓ_s} & & \\ \textcircled{\scriptsize ℓ_h} & \mathbf{val}\ \textcircled{\scriptsize ℓ_0} & i_0\ =\ 7 \\ & \mathbf{and}\ \textcircled{\scriptsize ℓ_1} & i_1\ =\ 13 \end{array}$$

  where the location ℓ_h, the *definition head*, represents the program location capturing the entire definition, for both of the identifiers i_0 and i_1. The location ℓ_s represents the program location immediately before ℓ_h, and ℓ_0 and ℓ_1 are the locations at the first and second definitions, respectively.

We observe the following facts:

$$\{\mathsf{Def}(i_0, \ell_0), \mathsf{Def}(i_1, \ell_1), \ell_h \lhd \ell_0, \ell_h \lhd \ell_1, \ell_h \prec \ell_s\}$$

meaning that identifiers i_0 and i_1 are visible to anyone who can see the location ℓ_h.

To complete the above **val** definition, we add facts $\ell_s \prec \ell_0$ and $\ell_s \prec \ell_1$. The bodies of the definitions of identifiers i_0 and i_1 now draw their own environments from the earlier location ℓ_s and therefore cannot see each other. If we want to model the above as a recursive definition (e.g., to model a **fun** or **val rec** definition), we instead add the edges $\ell_h \prec \ell_0$ and $\ell_h \prec \ell_1$ — the bodies of i_0 and i_1 can now reach the head of their own definition, ℓ_h, via $(\prec)$, and therefore (via $\ell_h \lhd \ell_0$ and $\ell_h \lhd \ell_1$) access each other's definitions. Each definition via Def thus has two edges — a $(\prec)$ edge to the location representing its surrounding environment, and a $(\lhd)$ edge to its associated definition head. Without this indirection, definitions in mutually recursive definition blocks with n definitions would need $O(n)$ edges to connect to all other definitions in the block, for a total of $O(n^2)$ edges.

- $\mathsf{PDef}(i, \ell)$: this relation records definitions of identifiers as parameters. Such definitions are not visible to subsequent definitions, only to the body of the definition (transitively reachable via $\prec$).
- $\mathsf{Head}(\ell_h)$: this relation describes a definition head, such as ℓ_h in our example above.
- $\mathsf{ValHead}(\ell_h)$ and $\mathsf{Val}(\ell_b)$: these two relations mark **val** definitions. Relation ValHead is a subset of Head and marks definition heads that are value definition heads; Val in turn marks the locations for each of the individual definitions underneath the definition head; in our earlier example we would mark ℓ_0 and ℓ_1 in this fashion, designating these locations as *definition branches*.
- $\mathsf{FunHead}(\ell_h)$ and $\mathsf{Fun}(\ell_b)$: these two relations are the equivalents to ValHead and Val, but for function definitions.

8.1.2 Inconsistency detection

The following conditions render Scope ML programs ill-formed:

- $\mathsf{Ambiguous}(\ell, n)$: the definition block at ℓ defines more than one identifier with the name n (**val** x = 1 **and** x = 2). Section 2.9 of the Revised Definition of Standard ML [MTHM97] dictates this condition.
- $\mathsf{VarCapture}(i, \ell_u, \ell_c)$: the use of the identifier i at location ℓ_u is subject to name capture at location ℓ_c.
- $\mathsf{VarNonReach}(i, \ell_u)$: identifier i, used at location ℓ_u, is not reachable from ℓ_u.
- $\mathsf{AmbiguousParam}(i, n)$: the function identified by i has multiple parameters with the same name n.

Figure 8.1 specifies the rules for deriving these conditions.
We use three auxiliary relations in this figure:

$$\frac{\ell \triangleleft \ell_1 \quad \ell \triangleleft \ell_2 \quad \mathsf{Def}(i_1, \ell_1) \quad \mathsf{Def}(i_2, \ell_2) \quad \neg(i_1 = i_2) \quad i_1 \overset{\mathsf{n}}{\mapsto} n \overset{\mathsf{n}}{\hookleftarrow} i_2}{\mathsf{Ambiguous}(\ell, n)}$$

$$\frac{\begin{array}{cccc} \neg\mathsf{InLimbo}(\ell_u) & \neg(i = i') & \Delta(i, \ell_\Delta) & \Delta(i', \ell'_\Delta) \\ \mathsf{Use}(i, \ell_u) & i \overset{\mathsf{n}}{\mapsto} n \overset{\mathsf{n}}{\hookleftarrow} i' & \ell_\Delta \ll \ell'_\Delta & \ell'_\Delta \ll \ell_u \end{array}}{\mathsf{VarCapture}(i, \ell_u, \ell'_\Delta)}$$

$$\frac{\mathsf{Use}(i, \ell_u) \quad \neg\mathsf{InLimbo}(\ell_u) \quad \neg\mathsf{Reach}(i, \ell_u)}{\mathsf{VarNonReach}(i, \ell_u)}$$

$$\frac{\mathsf{Def}(i, \ell) \quad \mathsf{PDef}(i_1, \ell) \quad \mathsf{PDef}(i_2, \ell) \quad i_1 \overset{\mathsf{n}}{\mapsto} n \overset{\mathsf{n}}{\hookleftarrow} i_2 \quad \neg(i_1 = i_2)}{\mathsf{AmbiguousParam}(i, n)}$$

$$\frac{\ell' \prec \ell}{\ell' \ll \ell} \qquad \frac{\Delta(i, \ell_\Delta) \quad \ell_\Delta \ll \ell}{\mathsf{Reach}(i, \ell)} \qquad \frac{\mathsf{PDef}(i, \ell)}{\Delta(i, \ell)}$$

$$\frac{\ell' \prec \ell \quad \ell'' \ll \ell'}{\ell'' \ll \ell} \qquad \frac{}{x = x} \qquad \frac{\ell_d \triangleleft \ell_b \quad \mathsf{Def}(i, \ell_b)}{\Delta(i, \ell_d)}$$

Figure 8.1: Datalog rules for deriving error conditions for Scope ML.

- $(\ll)$: Loc $\times$ Loc, which is the transitive closure of $(<)$.
- Δ : Id $\times$ Loc, where $\Delta(i, \ell)$ states that identifier i is defined at location ℓ, i.e., $\Delta = (\lhd \circ \mathsf{Def}) \cup \mathsf{PDef}$.
- Reach : Id $\times$ Loc, where $\mathsf{Reach}(i, \ell)$ means that identifier i is reachable from location ℓ by following $(\ll)$ to a location at which Δ applies, i.e., $\mathsf{Reach} = \Delta \circ \ll$.

In Figure 8.1, we see the inference rules for the various error conditions. Ignoring (for the time being) the use of InLimbo, let us consider the three rules for inferring inconsistencies in turn:

(1) $\mathsf{Ambiguous}(\ell, n)$ holds iff the definition head ℓ has two definition branches ℓ_1, ℓ_2 defining distinct identifiers i_1 and i_2 with the same name n.

(2) $\mathsf{VarCapture}(i, \ell_u, \ell_c)$ holds iff identifier i is used at location ℓ_u, and defined at ℓ_Δ, and if there is some other identifier i' with the same name n as i defined at location ℓ'_Δ, where $\ell_\Delta \ll \ell'_\Delta \ll \ell_u$. Informally, we can think of identifier i' as intervening when we try to determine the meaning of name n at location ℓ_u — looking up by name, we will find i', since it has masked i in the environment.

(3) $\mathsf{VarNonReach}(i, \ell_u)$ holds iff we use i at location ℓ_u, but cannot actually reach it from there.

(4) $\mathsf{AmbiguousParam}(i, n)$ holds iff identifier i has two distinct parameters, i_1 and i_2, which share the name n.

8.1.3 Transformations

Figure 8.2 lists a number of transformations for Scope ML. We omit the (straightforward) discussion of the AST transformation aspect of these transformations here and focus on their logical impact. We use the following notation:

$$t(x_1, \ldots, x_n) = \frac{D}{C}$$

means that the transformation t creates all properties listed in C and deletes all properties listed in D. The user-supplied variables $x_1, \ldots x_n$ are substituted into C and D.

- *rename* is the identifier renaming transformation we used previously. This transformation corresponds directly to *rename* refactorings, except that it has no preconditions. In particular, the program transformation part of *rename* simultaneously renames all defining and using occurrences of the identifier.

  This transformation uses two names: the new, user-supplied name n', and the old name n. n is not user-supplied: our Datalog prototype infers all free variables occurring in the deleted properties. Since each identifier has only one name at any given point in time, n is never ambiguous.

The remaining transformations support the task of relocating definitions, as well as eliminating unnecessary definitions.

- *elim-def* eliminates a definition of identifier i from a branch of a definition head ℓ_h; for example, in **val** a = 1 **and** b = 2, a single *elim-def* might eliminate the definition of either a or b.

  This rule has a precondition, namely that i is indeed defined at ℓ_b, i.e., $\mathsf{Def}(i, \ell_b)$. We have no special provisions for preconditions but achieve the same effect by listing this property among both constructed and deleted facts. Thus, the transformation only applies if $\mathsf{Def}(i, \ell_b)$ holds, but we retain $\mathsf{Def}(i, \ell_b)$.

  The model effect of eliminating the definition at ℓ_b from the program is that the ($\lhd$) relation from the definition head to the definition branch is severed, as is the ($\prec$) relation providing the environment for the definition body. This means that the body of the definition cannot access the outer environment, and that the identifier i is no longer reachable according to the language theory from Figure 8.1. Note that we do not actually destroy the part of the program model that we just disconnected: all facts about the definition body still exist.

  To be able to reconnect location ℓ_b to other parts of the AST later, we tag it as Limbo, which easily distinguishes it from locations still connected to the AST. Limbo serves a second purpose, as we shall see shortly.

- *intro-val* grafts a previously eliminated **val** definition branch ℓ_b (marked as Limbo) onto an arbitrary **val** definition block whose head is ℓ_h. Note the use of $\mathsf{ValHead}(\ell_h)$ to ensure that **val** definitions are not erroneously added to **fun** definition blocks.

  Together, *elim-def* and *intro-val* allow transforming e.g.

```
let val a = 1
in let val b = 2
   in a + b
end end
```

  into

```
let val a = 1
    and b = 2
in let in a + b
end end
```

- *intro-fun* is analogous to *intro-val*, except for **fun** definitions.

- *intro-val-hd* introduces a new **val** definition head between two Head locations (value or function definition heads). Recall that *intro-val* only allowed us to graft a definition branch onto an existing head; this is insufficient for moving a definition head. Taking our example from *intro-val*, we need to combine *elim-def*, *intro-val* and *intro-val-hd* to transform the program into

```
let val a = 1
    val b = 2
in let in a + b
end end
```

  where the definition of b now has its own (fresh) definition head. *intro-val-hd* lists an unbound variable ℓ' in its list of created facts: our Datalog system interprets such variables as fresh objects. This ℓ' is now precisely our new definition head.

- *intro-fun-hd* is the analogue of *intro-val*, except for function definition heads.

$$\mathsf{rename}(i,n) = \frac{i \overset{\mathsf{n}}{\mapsto} n}{i \overset{\mathsf{n}}{\mapsto} n'}$$

$$\mathsf{elim\text{-}def}(i,\ell_b) = \frac{\mathsf{Def}(i,\ell_b) \quad \ell_h \triangleleft \ell_b \quad \ell_h \prec \ell_b}{\mathsf{Def}(i,\ell_b) \quad \mathsf{Limbo}(\ell_b)}$$

$$\mathsf{intro\text{-}val}(\ell_b,\ell_h) = \frac{\begin{array}{cc} \mathsf{Val}(\ell_b) & \mathsf{Limbo}(\ell_b) \\ \mathsf{ValHead}(\ell_h) & \ell_s \prec \ell_h \end{array}}{\begin{array}{ccc} \mathsf{Val}(\ell_b) & \ell_d \triangleleft \ell_b & \ell_s \prec \ell_d \\ \mathsf{ValHead}(\ell_h) & & \ell_s \prec \ell_b \end{array}}$$

$$\mathsf{intro\text{-}fun}(\ell_b,\ell_h) = \frac{\mathsf{Fun}(\ell_b) \quad \mathsf{Limbo}(\ell_b) \quad \mathsf{FunHead}(\ell_h)}{\mathsf{Fun}(\ell_b) \quad \mathsf{FunHead}(\ell_h) \quad \ell_h \triangleleft \ell_b \quad \ell_h \prec \ell_b}$$

$$\mathsf{intro\text{-}val\text{-}hd}(\ell_\uparrow) = \frac{\mathsf{Head}(\ell_\uparrow) \quad \mathsf{Head}(\ell_\downarrow) \quad \ell_\uparrow \prec \ell_\downarrow}{\begin{array}{ccc} \mathsf{Head}(\ell') & \ell_\uparrow \prec \ell' & \mathsf{Head}(\ell_\uparrow) \\ \mathsf{ValHead}(\ell') & \ell' \prec \ell_\downarrow & \mathsf{Head}(\ell_\downarrow) \end{array}}$$

$$\mathsf{intro\text{-}fun\text{-}hd}(\ell_\uparrow) = \frac{\mathsf{Head}(\ell_\uparrow) \quad \mathsf{Head}(\ell_\downarrow) \quad \ell_\uparrow \prec \ell_\downarrow}{\begin{array}{ccc} \mathsf{Head}(\ell') & \ell_\uparrow \prec \ell' & \mathsf{Head}(\ell_\uparrow) \\ \mathsf{FunHead}(\ell') & \ell' \prec \ell_\downarrow & \mathsf{Head}(\ell_\downarrow) \end{array}}$$

Figure 8.2: Transformations for Scope ML.

The above definitions may lead to spurious error messages when definitions are deleted. As an example, consider the deletion of an unused definition **val** a = b. Since a is unused, its absence will not cause any inconsistencies. However, looking at our inconsistency inference rules, we notice that the use of b will cause a VarNonReach inconsistency: the use of the identifier of b remains part of the program model, even though we have presumably disconnected the concrete occurrence of b from the program. This runs contrary to intuition (and to our practical needs) which suggest that locations in limbo should not cause inconsistencies.

We address this problem by suppressing errors that arise in locations that are "in limbo" (as shown in Figure 8.1); the language theory rules for InLimbo are simply

$$\frac{\mathsf{Limbo}(\ell)}{\mathsf{InLimbo}(\ell)} \qquad \frac{\mathsf{Limbo}(\ell') \quad \ell' \ll \ell}{\mathsf{InLimbo}(\ell)}$$

Thus, any location that has an ancestor in limbo is also considered in limbo, and name captures and unreachable identifier uses for such locations are suppressed.

8.1.4 Automatic inconsistency recovery

Our Datalog prototype is unique in that it includes an automatic inconsistency recovery mechanism. Automatic inconsistency recovery is the process of finding sequences of transformations that eliminate any inconsistencies we have observed; we refer to such transformation sequences as *recovery plans*.

A naive depth-bounded search for recovery plans is impractical, as it

(1) is inefficient, in that it does not re-use information about how concrete inconsistencies came about,

(2) will find and report plans with unnecessary transformations simply because the transformations do not obstruct inconsistency recovery, and

(3) will find and report permutations of plans.

By exposing the program theory and model transformations to the planner, we can allow more intelligent planning approaches. The AI planning literature specifically focusses on transformations such as our model transformations (referred to as 'actions') [RN03, NFF+05], often extended with a general notion of preconditions. In the context of planning, our notion of a program theory corresponds to *domain axioms* [Gar00].

However, our initial experiments suggested that our Datalog infrastructure was not efficient enough to be of use even for the task of validating recovery plans (Appendix B). We thus did not explore this path very far, though we did observe a number of heuristics that may be of use for future systems and that we describe in the following.

Note that exposing program theory and model transformations gives rise to the *trace* of an observed inconsistency, i.e., to the sequence of facts that contributed, directly or indirectly, to the inconsistency; this 'trace' corresponds to the 'dynamic slice' of this fact from a program-slicing perspective, or to its 'proof', from a proof-theoretical one. We can unify ground facts from such traces with deleted abstract facts from our model transformations to find options for how such 'offending facts' might be eliminated.

Negated properties in traces are inherently harder to deal with than positive properties. Intuitively, when we encounter a negative (missing) property in a trace, we would like to create that property with some transformation to see whether this will eliminate the inconsistency we are considering.

Our system supports four heuristics for dealing with such existentially quantified variables:

(1) Use fresh objects. This technique is useful for dealing with *rename*, where we can always find a new, unused name.

(2) Re-create facts that contributed to the desired program model but are missing in the current program model. This technique is useful for recovering the original state of the program, prior to invalidation.

(3) Use objects (with matching sort) that occurred in the transformations leading up to the observed inconsistencies. We found this heuristic to be roughly as useful in producing recovery plans as #2, but less precise: fewer of the plans it found were useful.

(4) Consider objects from *similar* facts. When encountering variables in negated facts, we try to fill in these variables by unifying each negative fact with other facts occurring in our traces.

For each suggested transformation *t*, our system validates the transformation against the current/desired program models and program/language theories. If inconsistencies remain (or were newly added) after applying *t*, our system recursively searches for further transformations to recover, down to a user-specified depth.

8.2 Java prototype

We[1] implemented the Java prototype as a plugin for the Eclipse IDE (version 3.2.2). This prototype employed the same infrastructure that Eclipse's built-in refactorings use in order to make a comparison between the two approaches meaningful.

We first describe our prototype's program model, our consistency promises, and the PM steps it supports, followed by a discussion of our user interface and a demonstration of the flexibility of our system compared to traditional refactorings.

8.2.1 Program model

Our program model includes the results of name, Use-Def, and Def-Use analyses. For name analysis, we use Eclipse's built-in bindings mechanism to determine the declaration for each use of a name and store a mapping between names and declarations. For Def-Use and Use-Def chains, we calculate intra-procedural reaching definitions and similarly store a mapping between uses and definitions. We recompute the model when the program changes; our model equivalence test reports an inconsistency whenever the newly computed mappings do not match the original ones.

[1] The Java prototype's implementation stems from Devin Coughlin, to whom the author is indebted for his contribution.

8.2.2 Behaviour preservation

Our prototype tracks whether variables, classes, type variables and methods refer to the same entities as before metamorphosis. Since it is impossible in general to determine the precise dynamic type of an expression, our system uses static types to resolve dispatch; thus, we are sometimes inaccurate when determining whether two methods refer to the same piece of functionality before and during metamorphosis. Our system may therefore conservatively issue inconsistencies where there are none; the user can review such inconsistencies and override them as (potential) behavioural change.

We further track re-ordering among read and write operations in local variables, which can arise when we move code fragments via PM-Cut and PM-Paste (see below).

8.2.3 PM steps

We have focussed on implementing small, composable transformations that, when combined, can match and exceed the expressive power of common refactorings. To that end, our current prototype supports the following PM steps:

- *PM-Rename*: change the name of a type or variable and its uses. This step is similar to the 'Rename' refactoring except that it uses the current program model to link names to declarations. Unlike the 'Rename' refactoring, PM-Rename allows name changes that result in name captures or other inconsistencies. Since this step does not alter the *desired* program model, we lose no information when a renaming causes names to conflict.

- *PM-Split*: take a single assignment and convert it into a declaration and initialiser (such as 'x = y + 500;' $\longrightarrow$ "int x = y + 500;"). Unlike the 'Split Temporary' refactoring, this step does not introduce a new variable name for the declaration.

- *PM-Delegate*: replace a method call on implicit *this* with the same method call on another object or vice versa (e.g. 'bar()' $\leftrightarrow$ 'foo.bar()').

- *PM-Cut*: remove a statement, field, or method, along with its associated program model fragment, and place it in a clipboard. There is no analogue to PM-Cut in refactoring.

- *PM-Paste*: retrieve the statement, field, or method from the current clipboard and paste it and the program model fragment into a class or method body. There is no analogue to PM-Paste in refactoring.

Our PM steps act on both the AST and the program model. For example, PM-Split replaces an assignment AST node with a variable declaration node, but also updates the name mappings in the model so that each name that uses the definition now maps to the new declaration.

We have by no means implemented the complete set of useful PM steps. However, as Kiezun et al. point out [KFK07], the fraction of 'Rename' and 'Move' refactorings among all refactorings used in practice is very high, "perhaps as high as 90% of all refactorings". We chose to provide PM-Rename and PM-Cut/PM-Paste to support these refactorings. PM-Cut and PM-Paste also permit great flexibility in program evolution and demonstrate that program metamorphosis has utility beyond mere refactoring. PM-Delegate followed from PM-Cut/PM-Paste; it is very useful for clearing up inconsistencies when moving code between different classes and methods. We implemented PM-Split to provide support for the 'Split Temporary' refactoring and to illustrate that our notion of program models scales to program properties other than name analysis mappings.

8.2.4 User interface

Our prototype attempts to mirror the user interface workflow of Eclipse's refactorings as much as possible. The user selects a portion of program text in the main editor and then chooses a PM step from an Eclipse menu. This brings up a modal 'wizard' box that requests additional information (e.g., the new name in a PM-Rename step) where necessary. The user can then review a list of textual changes that the PM step will perform and can choose to apply or abort the step.

If the user chooses to apply the step, we ask Eclipse to perform these textual changes. We then recompute the model and compare it to the desired model, listing any differences as 'Problem Markers' in the Eclipse pane for syntax errors and warnings. The user may choose to accept any of these differences as a change in program behaviour using Eclipse's 'Quick Fix' interface and/or apply additional PM steps to resolve them.

In order to maintain our consistency guarantees, we must prevent the user from free-form editing the program text. While it may sometimes be possible to map arbitrary edits into appropriate program model updates (borrowing ideas from [TDX07]), we cannot expect such approaches to work in general. Consider a program with name capture: if the user writes a new statement referencing the captured name, it is unclear which declaration she means.

8.3 SML prototype

For our SML prototype, we used an attribute grammar specification (some examples of which we discussed in the previous chapter) to describe name and fixity analysis, as well as our name model.

We implemented the rest of our system by hand, first constructing an intermediate program model (Section 6.4.2) from the attribute grammar tool output and then adding all further program analyses and model attributes as needed.

We constructed our prototype to support all of Standard ML 97 [MTHM97], except for the obsolete `abstype` language feature and records. Since records in SML are labelled tuples, we can always simulate them in tuples; they thus do not present a conceptual challenge to our approach.

8.3.1 Program models

We have already extensively described our program models. In addition to our intermediate model, we used all of the following:

- A name model (Section 6.2.1.3),
- Term and effect models (Section 6.2.3),
- A congruence polarity model (Section 6.2.6), and

- External interface metamodels (Section 6.2.7) on the term model and on the congruence polarity model.

8.3.1.1 Extensions over the program models we described

Our term, effect, and congruence models support all aspects of SML supported by our system. To this end we also use a more precise form of exception support that distinguishes between different kinds of exceptions, and a more precise form of exception handling support that subtracts only exceptions that are fully handled by a given exception handler.

To support the Standard ML basis library and the effects it induces, we added an external specification of many of the basis library's modules. We then annotated effectful functions used in our benchmarks, such as printing and array access, to indicate their effects.

8.3.1.2 Congruence support

For our congruence cast support, our system automatically searches for congruence casts when trying to explain transitions between terms that are part of the term model. We omit this search for term transitions that occur in the effect model. Our underlying assumption here is that axioms only describe the term behaviour, not the effect behaviour. This is a heuristic assumption that does not always apply in practice; we expect that later revisions of our system may use two different kinds of axioms to allow users to pick between our current behaviour and the more general behaviour in which axioms specify both term and effect equivalence.

We list the default congruence specification for our system in Appendix E.

8.3.1.3 Term rewriting

Both for detecting term equivalence and for inferring congruences, we used a term rewriting engine to detect equivalences through an iterative deepening depth-first search. We bounded our search on a number of user-configurable parameters:

- Maximum number of congruences that we may use (introduce/eliminate) in one attempt at explaining a single term transition (set to 5 for our experiments).
- Maximum number of definition inlinings per term transition (set to 1 for our experiments).
- Maximum number of premises in rules that we may attempt to satisfy per term transition (set to 5 for our experiments).
- Maximum number of congruence-free rewrites per term transition (set to 3 for our experiments).

8.3.2 User interface

Our SML prototype runs either as a standalone command-line client, or as a background process connecting to an EMACS session. With EMACS, our prototype provided an alternative to the standard SML editing mode, with program transformations set to EMACS key bindings. As with our Datalog prototype, users would switch between regular and program metamorphosis mode, with program metamorphosis mode disabling free-form editing and interpreting most standard input as an attempt to rename the identifier at the selected program location.

Unlike our Datalog prototype, we used a separate EMACS frame to display inconsistencies and allow users to jump to or accept selected inconsistencies.

8.3.3 Transformations

Our program metamorphosis engine provides the following primitive transformation operations:

- **Rename**: renames an identifier. For this operation we provided two separate modes: 'global rename' (the default) which implicitly fixes up all references to the same identifier elsewhere to use the new name, and 'local rename', which only updates the name at the selected program location.
- **Introduce Definition**: introduces a **val** binding at the current program point, binding a single name to an arbitrary let-free expression. Our lack of support for let subexpressions is due to implementation artifacts and not conceptual.
- **Replace Expression**: replace one expression by another. Expressions here can take the same form as in **Introduce Definition**.
- **Delete Definition**: removes a definition at the current point in the program.
- **Introduce Assumption**: adds a natural equivalence specification (lemma, assumption, or full-fledged natural equivalence) to the program at the current program point.

Chapter 9

Evaluation of our prototypes

To understand the utility of our prototypes, we evaluated them by a number of criteria. First, we examined their transformative power (Section 9.1), comparing the transformations they could apply to established catalogues of transformations and (for the Java and SML models) concrete transformation tasks that were beyond the power of state-of-the-art refactoring engines.

For our Java and SML prototypes, we also analysed the strength of their behaviour-preservation guarantees (Section 9.2). For our Java prototype our evaluation was experimental, while we used a formal argument for our SML prototype. Furthermore, we determined their runtime performance for transformations (Section 9.3).

9.1 Expressiveness

For expressiveness, we compared our systems to the refactorings suggested by established catalogues of refactorings: Fowler's book on refactoring [FBB+99] and Thompson and Reinke's Catalogue of Functional Refactorings [TR01b], which to our understanding is the only such catalogue for functional languages.

Thompson and Reinke list 22 refactorings, all of which have duals. However, the first refactoring, *Renaming*, is identical to its dual; thus, they list a total of 43 refactorings. Of these, six are not applicable to SML (Refactoring #11 due to the lack of rank-2 polymorphism, Refactoring #14 due to the lack of a set comprehension mechanism, and #18 due to the lack of syntactic sugar for Monads).

9.1.1 Datalog prototype

We implemented the transformations listed in Figure 8.2 as well as transformations for new type aliases, substituting type identifiers by aliased type identifiers, and transformations to accept externally visible change. With these transformations, we found that our system subsumed the following seven refactorings from Thompson and Reinke's catalogue:

- *Renaming* (#1) is covered by the rename transformation.
- *Lifting* and *Demoting* (both #2) break down into elim-def and intro-val or intro-fun transformations, optionally including intro-val-hd and intro-fun-hd.
- *Naming a Type* (#3 and dual), which adds/removes a type definition and replaces type name occurrences by occurrences of definitionally equivalent type names. Our system provides two transformations *add-type-alias* and *substitute-alias* transformations which subsume this refactoring.

- *Migrate Functionality* (#10 and dual), which (in SML) moves definitions into or out of structures, is similar to *Lifting* or *Demoting*, except that the target locations are in opened structures.

9.1.2 Java prototype

Using the five PM steps supported by our system (cf. Section 8.2.3) we found that we can implement some refactorings completely, while offering partial support for others. Our prototype supports seven standard refactorings [FBB+99]: '*Rename*', '*Pull Up Method*', '*Pull Up Field*', '*Push Down method*', '*Push Down Field*', as well as '*Move Field*' and '*Split Temporary*.'

We currently have no facility for adding or removing classes; but if the user manually adds empty classes and uninitialised fields before beginning program metamorphosis and manually deletes other classes afterwards, we can support two additional refactorings, '*Tease Apart Inheritance*' (Section 5.1) and '*Extract Class*'. For many other refactorings, such as '*Move Method*', '*Inline Method*', or '*Replace Inheritance With Delegation*', our system can provide significant support.

Figure 9.1 describes the subsumptions in more detail. In that figure the **Kind** of subsumption indicates whether subsumption was **full** (meaning that our PM steps are fully sufficient), **near** (meaning that we needed to create empty classes and unused fields before and/or delete classes and fields afterwards), or **partial** (meaning that our support was still more limited, but nontrivial).

9.1.2.1 Moving methods

During our experimentation with program metamorphosis, we applied our system to the Functional Analyzer. While examining our source code, we frequently observed opportunities for relocating method definitions (often after method extraction); below we describe an abstracted example for 'PM-Move' similar to the ones we observed.

Suppose that we wish to move the method printValue from class A to class B, thereby performing the '*Move Method*' refactoring:

```
class A {
    public int getValue() { ... }
    public void printValue(PrintStream out) {
        out.println(getValue());
    }
}
class B {
    private A a;
    public void log() {
        a.printValue(System.out);
    }
}
```

We first PM-Cut the printValue() method in A and PM-Paste it into B. This results in an invalid program: the call to printValue() in log() and the call to getValue() in printValue() are applied to objects which do not respond to those methods.

```
class A {
    public int getValue() { ... }
}
class B {
   private A a;
   public void log() {
        a.printValue(System.out);
```

Name	Kind	PM steps	Description
Rename Field/ Method/Class	**full**	PM-Rename	Use PM-Rename.
Move Field	**full**	PM-Cut, PM-Paste	PM-Cut the field and PM-Paste into the target class.
Push Down Method	**full**	PM-Cut, PM-Paste	PM-Cut the member from the superclass and PM-Paste into the subclass or subclasses.
Push Down Field			
Pull Up Method	**full**	PM-Cut, PM-Paste	PM-Cut the member from one of the subclasses and PM-Paste into the superclass. PM-Cut identical members from other subclasses.
Pull Up Field			
Split Temporary	**full**	PM-Split, PM-Rename	Use PM-Split to convert an assignment to a declaration and then PM-Rename to rename either the original or the newly declared variable.
Tease Apart Inheritance	**near**	PM-Cut, PM-Paste, PM-Delegate	We discuss this in detail in Section 5.1.
Extract Class	**near**	PM-Cut, PM-Paste, PM-Delegate	Create the new class manually, then extract the class using PM-Cut/PM-Paste and PM-Delegate for all parts of class extraction that do not require manual coding (such as custom constructors or factory methods).
Move Method	**partial**	PM-Cut, PM-Paste, PM-Delegate	We discuss this in Section 9.1.2.1.
Inline Method	**partial**	PM-Cut, PM-Paste, PM-Rename	PM-Cut, PM-Paste, and PM-Rename can help the user with the inlining/name capture handling aspect, although we do not automatically substitute actual parameters.
Extract Method	**partial**	PM-Cut, PM-Paste	We discuss a similar transformation in Section 9.1.2.2
Inline Class	**partial**	PM-Cut, PM-Paste, PM-Delegate	PM-Cut fields and methods and PM-Paste into the absorbing class. Fix references with PM-Delegate.
Replace Inheritance with Delegation	**partial**	PM-Delegate	Use PM-Delegate to add the delegation.
Replace Delegation with Inheritance	**partial**	PM-Delegate	Use PM-Delegate to remove the delegation.
Remove Assignments to Parameters	**partial**	PM-Split, PM-Rename	Same approach as ‘Split Temporary’ for assignment expressions that are not r-values.

Figure 9.1: Java prototype: Refactoring subsumption table.

```
        }
        public void printValue(PrintStream out) {
             out.println(getValue());
        }
    }
```

We can fix this by applying two PM-Delegate steps: once to change ‘a.printValue()’ to just call ‘printValue()’ and once to change ‘getValue()’ to call ‘a.getValue().’ Our consistency checker validates that we are indeed referencing the correct methods, and so we arrive at the correct program:

```
    class A {
        public int getValue() { ... }
    }
    class B {
        private A a;
        public void log() {
             printValue(System.out);
        }
        public void printValue(PrintStream out) {
             out.println(a.getValue());
        }
    }
```

Our prototype does not support changing method parameters, so it is not powerful enough to fully support the ‘Move Method’ refactoring. We note, however, that Eclipse’s implementation is not complete either — it only supports moving a method to the class of one of its parameters or fields. Our implementation has no such restriction.

9.1.2.2 Gradual method body extraction

As part of our experimentation with the Functional Analyzer we encountered a particular problem that existing refactorings do not conveniently handle, namely method extraction in nontrivial cases. Consider the following example (simplified from Functional Analyzer for ease of exposition):

```
1    double comp() {
2        double x, y;
3        x = f();
4        y = g();
5        . . . //intervening code
6        return x / y;
7    }
```

Assume that we want to extract lines 3 and 4 into a separate method to clarify their meaning while leaving the intervening code in comp. There is no sequence of automatic refactorings that can accomplish this. We cannot apply ‘*Extract Method*’ directly on lines 3 and 4 since it does not support multiple return values. The best refactorings can do is

- ‘*Extract Local Variable*’ on ‘x / y,’
- *Manually* move the resultant local variable to immediately follow line 4, and
- select lines 2–5 and perform ‘*Extract Method*’.

The manual move, in particular, is troubling because if the value of x or y happens to change in the intervening code then the refactored program will exhibit different behaviour than the original.

This is an ideal case for program metamorphosis, but our prototype is not mature enough to handle it. With a more sophisticated PM implementation, however, we could first create a new method to extract to and then PM-Cut and PM-Paste the statements we care about:

```
double fraction() {
    double x, y;
    x = f();
    y = g();
    return x / y;
}
double comp() {
    . . . //intervening code
}
```

Note that if the intervening code happened to modify x or y, even our present prototype would now complain that there were missing definitions at 'x / y'. The program is ill-formed at this point: comp is missing a return statement. We could address this by introducing a local variable initialised with the result of calling 'fraction()' and adding a return statement yields that local variable:

```
double fraction() {
    double x, y;
    x = f();
    y = g();
    return x / y;
}
double comp() {
    double result  = fraction();
    . . . //intervening code
    return result;
}
```

As we have stated, our prototype is not powerful enough to perform this sequence of steps. We are missing a step to create a new method, a step to introduce a new local variable with the results of calling that method, and a step to add a return statement. We would also need to add a more sophisticated program model to ensure that the result of 'fraction()' is the same as 'x / y,' such as a variant of our term and effect model from Section 6.2.3.

9.1.3 SML prototype

With the transformations we described in Section 8.3.3, coupled with our SML prototype's behaviour-preservation guarantees, we support 13 of the following refactorings from Thompson and Reinke:

- *Renaming* (#1).
- *Lifting* and *Demoting* (#2 and dual) by introducing and replacing new but equivalent expressions.
- *Generalise* and *Inline* (#6 and dual) through appropriate expression replacements, since our term model's equivalence checks will inline definitions (up to a user-defined depth) to check for equivalence.
- *Migrate Functionality* (#10 and dual). Analogous to #2.

#	program	size	speedup MLton	SML/NJ	PolyML	optimisation idea
1	vector-concat	434	2.04	8.68	2.12	vector deforestation
2	smith-normal-form	22826	0.99	—[3]	1.0	list vectorisation
3	nucleic	155173	1.04	1.03	1.06	square comparisons
4	DLXSimulator'	90606	11.27	0.68	1.07	list vectorisation

Figure 9.2: SML Prototype: Experimental results for algorithmic optimisation. For each experiment we list the speedups (runtime before transformation divided by runtime after transformation) for three SML implementations. Program size is in bytes and does not include algorithmic specifications or utility libraries that we added to optimise the code, but it does include any changes we performed to eliminate record types.

- *Enlarge / Shrink function return type* (#15 and dual). We can support this transformation either through suitable congruence specifications or having the term equivalence checker show the equivalence of the program before and after the transformation through inlining.
- *Reorganise Arguments* (#16 and dual). Analogous to #15.
- *Add/Remove Arguments* (#17 and dual). Analogous to #15 and #16.

Note that changes that affect parameters are currently only possible for function abstractions via **fn** (e.g., by introducing or removing an **fn** abstraction in a function body or value definition). This means that we fully support functions defined via **val rec**, but not functions defined via **fun**. Our program model implements the same behaviour-preservation checks for either form of function definition.

The focus of our SML prototype was to facilitate algorithmic optimisation. To understand its utility for this particular form of program evolution, we used our tool to optimise four programs from the MLton benchmark suite[1] . Since our system does not yet handle record types, we avoided programs that used them; in one case (DLXSimulator) we manually eliminated uses of the record types from the program.

9.1.3.1 Methodology

To increase the generality of our results, we conducted our experiments using three different SML implementations: MLton [FW01], version 20070826 (MLton), SML of New Jersey [App92] 110.70 (SML/NJ), and PolyML [2] 5.2.1 (PolyML).

We ran each experiment ten times (averaging the speedup) on an Intel Core2 6600 2.4 GHz dual-core computer with 32+32 kiB of level 1 instruction+data cache and 4 MB of shared level 2 cache. To the best of our knowledge, none of the benchmarks exploited the second core. Our machine was configured for EM64T bit mode, with 2 GB of main memory, running Debian 4.1.2-24 on a Linux 2.6.26 kernel.

[1] http://mlton.org/Performance

[2] http://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-335/

[3] Runs did not complete within 24h.

9.1.3.2 Case Study #1: vector-concat

The first row of Figure 9.2 summarises our results for vector-concat. The vector-concat benchmark tests the efficiency of folding over a concatenated vector. We optimised this benchmark to first fold over one vector and then the other (this is an example of *deforestation* [Wad90b]).

To perform this optimisation we had to add two axioms (for deforestation) and accepted one change to an effect.

As a result of this optimisation, we obtained speedups of 2.04 (i.e., optimised program ran in less than half the time) to 8.68 (i.e., optimised program ran in less than 1/8th of the time).

9.1.3.3 Case Study #2: smith-normal-form

The 'smith-normal-form' benchmark computes a normal form of a matrix. For this benchmark we replaced the lists of lists by a vector of vectors which resulted in faster accesses at the cost of initialisation of the vectors.

To perform this optimisation we added two axioms. Unfortunately, we did not obtain a speedup for this program.

9.1.3.4 Case Study #3: nucleic

The 'nucleic' benchmark is a bioinformatics benchmark. For this benchmark, we replaced expressions of the form `sqrt(x) < 4` with expressions of the form `x < 4*4`, thus eliminating the square-root computation.

To perform this optimisation we had to add five axioms and one congruence.

Our optimisation resulted in a modest speedup, ranging from 1.03 to 1.06.

9.1.3.5 Case Study #4: DLXSimulator

The 'DLXSimulator' implements a CPU and cache simulation. As part of the simulation, the program accesses and updates the CPU's registers, which are stored in a custom ImmArray.immarray datatype. This datatype behaves like an SML vector but is implemented using lists. We replaced these lists with vectors.

To do this optimisation we had to accept 18 changes to effects and also made 17 changes to the congruence rules.

Our optimisation resulted in significant speedups on MLton (11-fold), modest speedup on PolyML (1.07) and slowdown on SML/NJ.

9.1.4 Transformative power: summary

Our two more recent prototypes for Java and SML expanded on the modest ambitions behind our initial Datalog prototype. For both of the newer systems we found that we could express transformations that eluded traditional refactoring systems, whilst subsuming transformations from existing refactoring catalogues. Note that for both cases, we subsumed more refactorings than we provided PM steps, which suggests that the primitives of program metamorphosis may be a more fundamental form of transformation than refactorings.

9.2 Behaviour-preservation guarantees

To estimate the behaviour-preservation guarantees of our systems, we ran a number of experiments with our Java prototype (Section 9.2.1). For our SML prototype, we instead provide a formal result over the strength of our behaviour-preservation guarantees (Section 9.2.4).

9.2.1 Java prototype

We opted to compare the safety of our PM steps with refactorings provided by an established refactoring system. Since PM steps are more fine-grained than refactorings, we constructed three standard refactorings out of the metamorphosis steps provided by our prototype. There are many ways to construct such refactorings in practice, if we include all possible automatic fixups. We chose to implement all of our refactorings in a very straightforward manner: transform, check for inconsistencies, and abort if there are any inconsistencies (simulating the effect of a refactoring precondition). While this does not exploit the inherent flexibility of program metamorphosis, it is sufficient to address our principal experimental concern, safety.

9.2.2 Experimental Setup

For our experiments, we paired our manually constructed refactorings with refactoring built into Eclipse 3.2.2 (since our system was developed for the Eclipse 3.2 infrastructure). Our refactorings were as follows:

- **Rename**. Our 'Rename' refactoring simply performs a PM-Rename step, but does not attempt to avoid or resolve any name capture. We configured Eclipse's Rename to rename all other relevant identifiers, including identifiers of overriding and overridden methods in super- and subclasses, which our renaming does not do implicitly. Eclipse also provides a feature that will rename occurrences of a class name in a string or external text file. This option is meant to address uses of reflection, wherein Java may instantiate a class, invoke a method, or read from or write to a variable designated by a string value. Since this mechanism is unsafe in practice, we left it disabled.
- **Pull Up Field**. Our 'Pull Up Field' refactoring moves a field to a superclass (PM-Cut followed by PM-Paste), then iterates over all subclasses of the target class to identify fields of the same name. For each such field it tests if the initialiser is identical to the initialiser of the initially selected field, and, if so, deletes the field in the subclass (per PM-Cut).

  For Eclipse's 'Pull Up Field', we instructed Eclipse to also pull up dependent methods and fields, if necessary (in practice, this should only be needed if those entities are used in the field's initialiser.)
- **Pull Up Method**. Our Pull Up Method refactoring implementation is analogous to Pull Up Field, except that we also determine all methods and fields transitively referenced in the method and pull those up afterwards.

  We configured Eclipse's Pull Up Method to also move all dependent entities.

We then instructed our system to randomly locate opportunities for applying such refactorings in a given program. Our mechanisms for choosing such opportunities were as follows:

- **Rename**: For every 'Rename', we identified a possibly renameable entity (corresponding to a 'SimpleName', in Eclipse JDT nomenclature) anywhere in the program. We skipped package names because of limitations of our testing infrastructure, but included class names and names of entities external to the program (such as the 'toString()' in java.lang.Object).

  We then decided a new name as follows: with a probability of 0.5 we chose a fresh name, otherwise we chose a random name from the same compilation unit. These names were chosen the same way that renameable entities were chosen; in particular, names that occur frequently in a class had a higher probability of being chosen. If the new name was identical to the original name, we instead chose a fresh name.

- **Pull Up**: For pulling up, we identified pairs of types (interfaces, abstract classes, concrete classes) in a nontrivial supertype relationship (i.e., the classes were not identical) together with a method or field that could be moved from one type to the other, as required by the specific refactoring.

To test the correctness of a refactoring, we tested for whether the refactoring aborted, succeeded, or failed. We say that a refactoring *aborted* if the refactoring indicated that it was not applicable / would change behaviour. In traditional refactoring terms, this is usually expressed as the precondition failing. We say that a refactoring *succeeded* if the refactoring applied, and the program was both statically well-formed and dynamically behaved the same as before, as far as we could tell (see below). We say that a refactoring *failed* if the refactoring applied (i.e., the precondition did not fail) but the resulting program was statically ill-formed or did not preserve its dynamic behaviour.

To determine whether dynamic behaviour had changed, we ran the unit test suite shipped with the programs in question. If any unit test failed, we assumed that dynamic behaviour had changed in an unintended way and that the refactoring had therefore failed.

We also automatically asserted that all non–aborted transformations had indeed modified the program and manually sampled the results to ensure that the transformations were reasonably close to our expectations.

Our specific approaches for determining refactoring results were as follows:

- **Eclipse refactoring**: For Eclipse's refactoring, we attempted to apply the refactoring (using the refactoring scripting interface) atomically. If the attempt failed (usually because a precondition failed), we marked the refactoring as *aborted*. Otherwise we ran Eclipse's own static checks on the program and any unit tests. If either the static checks or the unit tests failed, the refactoring *failed*, otherwise it *succeeded*.

- **Program metamorphosis**: For our own refactorings, we applied all relevant transformations (usually several) in sequence, disregarding any inconsistencies until the end. After we had finished transforming, we ran our own inconsistency checks as well as Eclipse's static checks. If either indicated an error or inconsistency, we *aborted*. Otherwise we ran the unit tests to determine whether the refactoring had *failed* or *succeeded*.

Note that we interpreted the results of Eclipse's own correctness checks differently for program metamorphosis and traditional refactoring. This reflects the program metamorphosis philosophy and highlights an advantage of our approach: by definition, a traditional refactoring must preserve behaviour if its preconditions trigger — in particular, it must produce a well-formed program. Program metamorphosis, on the other hand, need only be able to determine whether the program is well-formed or not *after the fact*. As we observed with Eclipse, this allows us to exploit traditional IDE correctness checks to augment our own checks for program model equivalence (Section 8.2.1).

9.2.3 Results

We ran our experiments against the following programs:

- Functional Analyzer [MSHD07], a flexible tool for fast analysis of trace information and similar numerical data, developed at the University of Colorado (7714 loc[4]).
- Apache Commons: Discovery 0.4, a library for detecting and managing plugins, developed by the free Apache Commons project (2543 loc).
- Apache Commons: Validator 1.3.1, a general-purpose validation library for structured data, particularly XML (8874 loc).
- Apache Commons: Chain 0.4, a chain-of-responsibility implementation, again part of the Apache Commons project (8010 loc).
- Apache Commons: Digester 1.8, a configurable XML configuration file interpreter (12342 loc).

We chose the above programs by availability and presence of substantial unit test suites.

For each experiment, we configured our system to perform 200 random transformations for each refactoring. Table 9.1 summarises our results.

As we can see from our results, our (fairly straightforwardly) PM-scripted refactorings are competitive with Eclipse's. In the majority of the cases we tested, both systems behaved equivalently. Where they didn't, the differences were mostly due to Eclipse being more flexible by providing additional fixups or Eclipse being less conservative (particularly when pulling up) and thus being simultaneously more flexible and more error-prone. For 'Pull Up', our primitive dependency analysis was sometimes fooled, most commonly by `this` references, resulting in additional aborts. In all instances that we observed, a human programmer, driven by our inconsistencies, would have been able to identify and rectify the situation straightforwardly. In other instances, less-than-ideal interfacing between our module and the Eclipse parser prevented our prototype from matching up code from before and after a transformation (particularly in Rename). With respect to the focus of our tests, we observed that PM-scripted refactorings were safer than Eclipse's traditional refactorings: averaging over all of our tests, the cases in which the PM-scripted refactorings failed and Eclipse's refactorings succeeded or aborted made up 0.1%, while the cases in which Eclipse's refactorings failed and the PM-scripted refactorings aborted or succeeded made up 24.4% of all tests.

- **Pull Up Field.** Pulling up, Eclipse attempts to merge fields from all subclasses, whether or not those fields have the same initialisers. This frequently introduces bugs, not all of which are caught by unit tests. If the common fields' types mismatch, Eclipse aborts, while our simple pull-up heuristic skips the fields if their initialisers differ, accounting

[4] Lines of non-comment non-whitespace source code, computed with `sloccount`.

Refactoring	Identical abort	Identical success	Identical failure	Identical total	More Flexible PM	More Flexible Eclipse	More Failures PM	More Failures Eclipse
Pull Up Field								
Functional Analyzer	9.0%	4.0%	0.0%	13.0%	2.0%	7.0%	0.0%	80.0%
Commons Discovery	13.5%	9.0%	0.0%	22.5%	0.0%	37.0%	0.0%	40.5%
Commons Validator	31.5%	4.5%	0.0%	36.0%	18.5%	32.5%	0.0%	31.5%
Commons Chain	16.5%	0.0%	0.0%	16.5%	1.5%	40.0%	0.0%	43.5%
Commons Digester	3.0%	19.0%	0.0%	22.0%	0.0%	43.0%	0.0%	35.0%
average	14.7%	7.3%	0.0%	22.0%	4.4%	31.9%	0.0%	46.1%
Pull Up Method								
Functional Analyzer	55.0%	3.0%	0.0%	58.0%	0.0%	20.5%	0.0%	21.5%
Commons Discovery	51.5%	0.0%	0.0%	51.5%	0.0%	20.0%	0.0%	28.5%
Commons Validator	46.0%	29.0%	7.5%	82.5%	0.0%	9.0%	0.0%	8.5%
Commons Chain	51.5%	0.5%	0.0%	52.0%	1.5%	5.0%	0.0%	42.5%
Commons Chain	46.0%	4.5%	2.5%	53.0%	0.0%	19.5%	0.0%	27.5%
average	50.0%	7.4%	2.0%	59.4%	0.3%	14.8%	0.0%	25.7%
Rename								
Functional Analyzer	17.0%	60.5%	0.5%	78.0%	0.0%	21.5%	0.0%	0.5%
Commons Discovery	29.0%	59.5%	2.0%	90.5%	0.0%	9.0%	0.0%	0.5%
Commons Validator	30.5%	60.5%	1.5%	92.5%	0.5%	5.5%	0.5%	1.5%
Commons Chain	43.0%	50.0%	1.0%	94.0%	0.5%	2.0%	0.5%	3.0%
Commons Chain	29.0%	59.0%	1.0%	89.0%	0.5%	9.5%	0.5%	1.0%
average	29.7%	57.9%	1.2%	88.8%	0.3%	9.5%	0.3%	1.3%

Table 9.1: Java prototype: comparison of behaviour-preservation guarantees to Eclipse. This table lists the benchmarking results for Eclipse's refactoring suite (Eclipse) and refactoring scripted from program metamorphosis steps (PM). **Identical** identifies cases in which both tools behaved equivalently. **More Flexible** identifies cases in which one tool permitted a transformation while the other tool aborted that transformation. **More Failures** identifies cases in which one tool caused behavioural change. Note that **Identical(total)**, **More Flexible** and **More Failures** sometimes add up to more than 100% in cases where both tools performed the transformation but one tool produced an incorrect result (we counted this as the correct tool being both safer and more flexible). Considering cases where Eclipse failed, this accounts for all of the cases in which PM was more flexible in 'Pull Up Field', as well as for 1% of the 'Pull Up Method' cases in the Commons Chain. Considering cases where PM-scripted refactoring failed, this accounted for two cases in 'Rename' (cf. our discussion).

for a few cases in which our PM-scripted refactoring is more flexible. In other cases, Eclipse implicitly changed field visibility (from private to protected) if needed, which was not part of our PM scripting.

- **Pull Up Method.** For 'Pull Up Method', both refactoring implementations failed consistently when pulling up unit test methods into superclasses for which not all subclasses satisfied the test, in the Commons Validator. When pulling up methods, Eclipse again suffered from its implicit merging of fields when pulling up dependent entities, while our PM-scripted refactoring's refusal to implicitly change visibility accounted for much of its lack of flexibility. Eclipse's 'Pull Up Method' further interprets requests to pull up a method into an interface as requests to add a method declaration of matching signature to the interface, again adding to its flexibility (an actual 'Pull Up Method' into an interface is only possible for abstract methods).

- **Rename.** For renaming, our system primarily suffered from two limitations: first, our prototype will not refactor constructors in some cases, and secondly, we do not enforce the override status of overriding methods in subclasses.

  Refactoring constructors requires renaming the class and all related constructors. A limitation of the Eclipse parser that we have not yet addressed sometimes prohibits this in classes with multiple constructors; this issue accounts for 3% of the aborted rename attempts (total) in the Functional Analyzer, 10.5% in Commons Discovery, 13% in the Commons Validator, 16.5% in the Commons Chain package, and 12% in the Commons Digester.

  Note that these failures also account for some of its increased safety in the presence of reflection. Since reflection allows classes to be looked up by names read from external files, it is notoriously hard to support in any kind of refactoring process. (All of our test cases utilise reflection to some extent.)

  Another current limitation is that our inconsistency checks do not enforce the overriding status of methods when renaming methods of a subclass. In the presence of an `@Override` annotation, this usually leads to static errors, but in two cases it allowed the PM-scripted refactoring to introduce a dynamic failure. We expect to extend our program model to add either explicit 'method-X-overrides-method-Y' information or global value numbering to increase the strength of our correctness promises overall.

We also experimented with 'Push Down Method' and 'Push Down Field'. Due to an unresolved issue in our prototype, our 'Push Down' operations are currently overly conservative when pushing to multiple subclasses: copying (rather than cutting and pasting) generates 'fresh' methods and fields, resulting in spurious inconsistency warnings that cannot be accepted as behavioural change. Conversely, Eclipse's 'Push Down' refactoring cannot be constrained to push down to one particular subclass: instead, it always pushes down to all immediate subclasses, though users can interactively choose to suppress parts of the textual diff after the refactoring has terminated. We could thus not directly compare the two sets of functionality, though we have no reason to assume that a corrected PM-scripted 'Push Down' would ultimately exhibit correctness or performance characteristics different from the PM-scripted 'Pull Up'.

While our results overall indicate that our scripted refactorings are less flexible than Eclipse's refactorings, we note the following:

- Our prototype is, on average, safer than Eclipse's refactorings.

- Our prototype permits us to quickly script refactorings that are as flexible as Eclipse's refactorings in most of the cases we examined, without including any automated fixups or complex analyses as part of the scripting.

9.2.4 SML prototype

Our SML prototype is fully behaviour-preserving by default, with the following exceptions:

- Explicit behaviour evolution
- Incorrect axioms in our congruence specification language
- Termination

Specifically, our system cannot detect when we introduce a value definition whose evaluation may never terminate.

We prove this for SiML, the restricted subset of SML that we introduced in Section 6.2.3, in Appendix D. Note that this almost solves our last challenge, Challenge #5: our solution only falls short of handling termination behaviour preservation.

9.2.5 Safety guarantees: summary

Our experimental and theoretical results show that program metamorphosis can provide safety guarantees that are on par with or better than state-of-the-art refactoring systems.

9.3 Runtime performance

Lastly, we examine how responsive program metamorphosis is in practice.

9.3.1 Java prototype

One goal of our Java prototype is to examine whether program metamorphosis is practical to implement and useful for evolving real-world programs. Here, we evaluate our prototype in terms of code size and resource consumption.

The main component of the memory cost for program metamorphosis is the need to keep an AST of the entire program in memory at all times. Our prototype requires two copies of the full AST in memory during equivalence checking. Using the Eclipse JDT's built-in memory queries, we have determined that a single instance of the Functional Analyzer AST requires approximately 4MB of memory, which we consider to be acceptable on modern machines.

Our prototype (excluding unit tests) consists of 3829 lines of Java across 53 files and relies significantly on Eclipse's infrastructure to perform program analysis and to interact with the user. This shows that a useful set of PM steps can be implemented in a relatively small amount of code and that PM can be compatible with existing program analysis frameworks and program evolution tools. For this reason, we have favoured ease of implementation and tight integration with Eclipse over speed. For example, we used the JDT's built-in name analysis even though it requires re-parsing to get updated analysis. We could reduce our runtime overhead by comparing only altered parts of the program and recomputing program models lazily. This would decrease execution times and memory usage at the cost of added complexity.

To ensure that our prototype is practical for interactive use, we measured execution times for the behaviour-preservation tests from Section 9.2.1. We summarise these results in Table 9.2. All experiments were run on a 2.4GHz Intel Core 2 Quad with 4GB of RAM, running Java 1.6.0_03-b05 on Ubuntu 7.10 (Gutsy Gibbon) with Linux 2.6.24.

For program metamorphosis, the execution time is the sum of the execution times of all intermediate steps. The **total** line gives the overall minimum, maximum, and average of the averages (as summarised per program).

Our unoptimised prototype executes most transformation steps well within the time limits of what we can expect from an interactive tool. While some transformations may take more than two seconds to complete overall, note that both of our pull-up refactorings are multi-step transformations in program metamorphosis; except for two Renames (one in the Functional Analyzer and one in the Commons Digester), no individual transformation execution time was more than 2.5s per PM step (including re-parsing and AST re-matching for the entire program after each step).

9.3.2 SML prototype

For our SML prototype, we measured the amount of time that it took our system to re-analyse and compare the program models in our four case studies. Using the same machine setup as for measuring the speedup, we ran each transformation ten times. For this purpose, we compiled our tool with MLton 20070826. For each of our tests, we configured the system to be equipped with all axioms relevant for our respective analyses.

Table 9.3 summarises the time needed for individual transformation steps for each of our benchmarks. For smith-normal-form and nucleic, our timings suggest that our system configuration is too slow for interactive use. To understand the causes for this time consumption, we analysed time consumption in more detail.

Table 9.4 summarises our breakdown. Except for smith-normal-form, which encounters an unidentified performance bug during congruence polarity type checking, most benchmarks spend the vast majority of their runtime during initial name analysis, suggesting that our attribute grammar framework may need an overhaul.

9.4 Discussion

As the experiments with our prototypes show, program metamorphosis captures both the common refactoring tasks and extended transformation tasks that elude state-of-the-art refactoring engines, whilst offering equal or better safety guarantees. By contrast, our initial runtime performance results are not yet conclusive: while the runtime performance results for our prototypes suggest that our approach is sufficiently efficient for small to medium-sized programs, it is unclear whether or how program metamorphosis can be scaled to programs of the size of what current refactoring tools can handle, or what the appropriate techniques for doing so might be.

In principle, a program metamorphosis system may need more memory than a refactoring system, since it must maintain both a current and a desired program model. In practice, recent work by Schäfer, Ekman and de Moor [SEdM08b] suggests that practical implementations of traditional refactoring may have to take an implementation approach similar to ours.

Examining how program metamorphosis can be scaled to larger programs is then one of the possible future directions for this research (Chapter 10).

		Eclipse			PM		
Refactoring	**Program**	**min**	**avg**	**max**	**min**	**avg**	**max**
Pull Up Field	Functional Analyzer	0.11s	0.32s	1.15s	2.05s	2.29s	2.81s
	Commons Discovery	0.13s	0.24s	0.52s	0.86s	0.97s	1.37s
	Commons Validator	0.18s	0.45s	0.96s	2.70s	3.07s	4.55s
	Commons Chains	0.36s	0.47s	0.83s	2.39s	2.49s	2.64s
	Commons Digester	0.15s	0.31s	1.51s	3.09s	3.44s	7.87s
	total	0.11s	0.36s	1.51s	0.86s	2.45s	7.87s
Pull Up Method	Functional Analyzer	0.12s	0.32s	0.83s	1.90s	2.55s	3.35s
	Commons Discovery	0.14s	0.32s	2.95s	**n/a**	**n/a**	**n/a**
	Commons Validator	0.22s	0.47s	1.93s	2.73s	3.79s	11.65s
	Commons Chains	0.39s	0.64s	1.74s	2.18s	2.38s	2.59s
	Commons Digester	0.19s	0.55s	1.62s	3.09s	4.83s	10.43s
	total	0.12s	0.46s	2.95s	1.90s	3.39s	11.65s
Rename	Functional Analyzer	0.09s	0.34s	2.71s	1.10s	1.36s	2.92s
	Commons Discovery	0.02s	0.15s	0.85s	0.43s	0.53s	1.53s
	Commons Validator	0.06s	0.20s	0.69s	1.46s	1.71s	1.98s
	Commons Chains	0.06s	0.26s	0.98s	1.14s	1.35s	2.12s
	Commons Digester	0.12s	0.28s	1.66s	1.72s	1.95s	2.58s
	total	0.02s	0.25s	2.71s	0.43s	1.38s	2.92s

Table 9.2: Java prototype: refactoring execution times. Minimum, average, and maximum refactoring execution times, for both Eclipse's built-in refactorings and program metamorphosis.

			model analysis time		
#	**benchmark**	**size**	**minimum**	**average**	**maximum**
1	vector-concat	434	0.108s	0.110s	0.116s
2	smith-normal-form	22826	26.574s	29.454s	33.306s
3	nucleic	155173	23.593s	23.693s	23.793s
4	DLXSimulator'	90606	6.796s	7.212s	7.904s

Table 9.3: SML prototype: transformation step time, totals.

pass	vector-concat	smith-NF	nucleic	DLXSim
Attribute grammar	52.17%	12.97%	75.74%	76.78%
Name model check	0.0%	0.0%	0.0%	0.0%
Type inference	1.09%	0.37%	13.61%	3.51%
Effect inference	2.9%	0.51%	7.05%	12.99%
Term model	0.36%	0.06%	0.33%	0.38%
Precise effect model	0.36%	0.07%	0.44%	0.53%
Term/Effect inconsistencies	42.75%	6.58%	0.27%	2.47%
Congruence model and checks	0.36%	79.45%	2.56%	3.35%

Table 9.4: SML prototype, benchmarks: relative time, broken up by program analysis task. Here, smith-NF is the smith-normal-form benchmark, and DLXSim is the DLXSimulator' benchmark.

Chapter 10

Outlook

While we have established basic results about the utility of program metamorphosis, there are many questions that we left unanswered in this work. In this chapter, we sketch some of the possible avenues for future research on the matter of program metamorphosis.

10.1 Scaling program metamorphosis

Perhaps the most pressing question is that of scalability. While our experimental results showed that our program metamorphosis prototypes were efficient enough for moderately-sized programs, it is unclear how well our approach would compare to traditional refactoring when applied to larger programs.

Challenges. Research in this area would involve building a new program metamorphosis system or optimising our existing prototypes to explore techniques for scaling program metamorphosis, and comparing the memory footprint and computational overhead of program metamorphosis with that of refactoring.

Initial ideas. Our attribute grammar approach (Chapter 7) may provide a basis for such optimisation. By abstracting from the concrete program metamorphosis system implementation, this technique may allow us to more easily compare systems that follow the same language specification. Specifically, the explicit data dependencies that we establish in attribute grammars lend themselves well towards *partial* invalidation and program model recomputation, though this may require us to first overcome the remaining inefficiencies in our attribute grammar implementation (Section 9.3.2).

Another technique that may be useful irrespectively of the concrete program metamorphosis infrastructure would be *lazy model instantiation*: instead of computing precise program models for all parts of the program, can we compute abstract models that we only concretise on demand?

10.2 Scripting PM steps

In our implementations and tests, we applied each PM step by hand. To properly subsume refactoring, program metamorphosis could benefit from mechanisms for scripting traditional refactoring steps from PM steps.

Challenges. This task would involve language design and implementation for a language that operates on programs, PM steps, and inconsistencies. Ideally, the language would have guaranteed termination properties (i.e., be sub-Turing complete) and allow us to show that each transformation will be able to fully handle all of the inconsistencies it might trigger.

Initial ideas. A PM step scripting language could come about in at least two guises: firstly, as a *proactive* language that encodes each refactoring as an initial sequence of PM steps followed by handlers for all anticipated kinds of inconsistencies, or secondly as a *reactive* language that again implements each refactoring first as a sequence of PM steps, but then backs away to let a hierarchy of *universal* (i.e., refactoring-independent) recovery mechanisms restore program validity. Both ideas are closely related to recovery planning, our next possible venue.

10.3 Recovery planning

While we found in our experiments that program metamorphosis is 'manageable,' in the sense that our inconsistency reports were sufficiently precise and meaningful for us to understand and resolve them, other programmers may disagree. Furthermore, future transformations and program models may be less transparent than the ones we used, but even now, complex inconsistencies (such as the ones we encountered with our congruence polarity model, Section 6.2.6) sometimes require a tediously large number of 'obvious' transformations.

While our initial results in recovery planning (Appendix B) were discouraging from the perspective of run-time performance, we did not find that this approach is intrinsically unfeasible.

Challenges. This avenue of research would focus on devising heuristics and other techniques for speeding up recovery planning and producing 'more useful recovery plans.'

Initial ideas. Two recent advances may make this approach more practical: first, modern Datalog evaluators may be significantly more efficient than our implementation whilst avoiding the fine-tuning needs of BDD-based implementations [BS09], possibly re-enabling Datalog as a viable venue for implementing program metamorphosis. Second, we note that the attribute computations in our attribute grammar approach (Chapter 7) are similar in style and expressive power to those of Datalog. Thus, our existing ideas for recovery planning (Section 8.1.4) may translate directly to attribute grammars; alternatively, we may find it possible to use our attribute grammar approach to construct Datalog models[1] .

10.4 Formal verification

As noted by Schäfer, Ekman and de Moor [SEdM08a], refactoring is a prime candidate for machine verification. Their proposal of simplifying this task by splitting refactoring into smaller steps to be verified individually, and of using models of the language that do not capture all of the language semantics precisely mirror some of the insights in program metamorphosis, except for a different task. Our work might thus provide a suitable basis for exploring their ideas.

Challenges. One task for this challenge would be identifying, formalising, and establishing relevant properties about suitable metatheories, another would be to establish the soundness of a given refactoring system.

Initial ideas. Again our attribute-grammar based approach (Chapter 7) may provide a basis for simplifying this challenge, since it provides a natural way for specifying static semantics and could be extended towards specifying dynamic semantics. With that approach, this challenge could be tackled in at least two ways: either by using such dynamic semantics to prove properties about properties in the fashion suggested by Schäfer et al., or by using the

[1] Our system includes a prototype Datalog backend for this precise reason, though the backend does not support some of the more recent extensions, such as type inference.

dynamic safety guarantees of program metamorphosis as a frame axiom and instead focussing the proof work on showing that the refactoring will be ‘complete,’ i.e., always eliminate all inconsistencies.

10.5 Tools for building PM tools

We implemented our three program metamorphosis prototypes mostly in custom hand-written code. As the sole exception, we generated the name analysis frontend for our SML prototype from an attribute grammar specification (Chapter 7). It may be feasible to push this approach further, in the spirit of existing compiler generators (Section 4.1.3), to build a general framework for deriving program metamorphosis systems from a language specification. This would be a significant undertaking with many individual challenges, but it may offer new solutions to the challenges we outlined in Sections 10.1, 10.4 and 10.3.

Challenges. This project would have to find answers to a large number of questions:

- How can we represent programs and recompute program models efficiently?
- What forms of specification are most suitable for static language semantics? Our current attribute grammar approach has many advantages, but is insufficient e.g. for fixed point computations.
- What forms of specification would be most suitable to give the dynamic language semantics, if we wish to derive program models from them?
- Can we *automatically* derive program models from such specifications? If not, what user support do we need to derive program models that guarantee full behaviour preservation?
- Can we *automatically* construct PM steps from abstract user descriptions?
- Can we scale this approach to safe full-fledged programming languages such as Modula-3, Standard ML, or Java?
- Can we scale this approach even to unsafe programming languages, such as C?

Initial ideas. One possible approach to tackle these questions would be to examine the manually implemented parts from our SML prototype and to try to devise a system general enough to capture both them and likely needs for other languages. Another would be to devise such a system ‘from the ground up,’ for languages with very limited scope, such as our language L0 (Appendix F).

10.6 User studies

We have not yet evaluated program metamorphosis through user studies. Thus, while we have shown that *in principle* program metamorphosis offers better behaviour-preservation guarantees and more flexibility than traditional refactoring, we do not yet know whether programmers can exploit these advantages *in practice*.

Challenges. This project would focus on extending and polishing our Java prototype, or implementing a new prototype for a programming language with an existing, mature refactoring engine, and setting up user studies to examine the practicality of our approach.

Initial ideas. For a direct comparison between program metamorphosis and refactoring, both techniques should be used for the same tasks. However, this project could also study whether programmers are able to exploit the behaviour-evolving mechanisms that program metamorphosis offers over refactoring, and whether they find these mechanisms to be useful.

10.7 Applying program metamorphosis to other tasks

In addition of extending and measuring the utility of program metamorphosis for its core task of software evolution, we may also be able to utilise it as a component for other systems.

10.7.1 Dynamic software updating

Dynamic software updating [SHM09] is the task of updating a running program without interrupting the services it provides. Part of the challenge in dynamic software updating is detecting how we should migrate program state from one implementation of a program to another.

If the programmers use program metamorphosis instead of manual program evolution, our PM steps and congruence casts may provide important clues to understanding the meaning of certain changes.

Challenges. The principal challenge for this project would be to classify the kinds of changes that program metamorphosis can understand and to match them to the kinds of changes supported by dynamic software updating, extending a given program metamorphosis tool as needed. These extensions might both add new kinds of transformations (in areas supported by dynamic software updating), new kinds of program models and analyses (to understand changes in terms that dynamic software updating can handle), and changes to PM steps to allow user annotations.

Initial ideas. Our congruence specifications allow us to define e.g. when and how two types behave equivalently. However, these specifications need not require a direct translation from one to the other. When we do have this information, we can replace one datatype by the other, but when we do not, we may still be able to *emulate* the new type while running an old one through a wrapper object of compatible type. If we combine this approach with dynamic class-loading, it can be almost fully transparent to the running program (except for uses of reflection), similarly to the approach taken by Henkel for debugging algebraic specifications in Java [HRD08].

10.7.2 Gradual programming

Gradual programming is a process for developing program and programming language simultaneously.

We argue that program metamorphosis may be a useful component of a gradual programming system: in program metamorphosis, we abstract program behaviour into a program model. If this program model captures *all* program semantics, we can reduce programming languages and syntactic program representations to mere projections of the program's essence, incorporated in the model.

It remains to be seen whether we can make this approach practical, just like gradual programming itself.

Chapter 11

Contributions

In this dissertation, we explained how pre-existing techniques for program development fail to provide support for many of the needs of software evolution. We then explained how we could overcome many of the limitations of these techniques.

By decoupling safety guarantees from transformations and establishing a 'desired program model,' capturing and enforcing expected program behaviour, we allowed our users to transform programs more flexibly than in existing refactoring tools. By allowing changes to this desired program model, we then allowed our users to deliberately evolve program behaviour. Finally, by allowing users to describe classes of congruent datatypes, we allowed them to express sophisticated algorithmic optimisations.

To evaluate the practicality of this approach, we implemented three interactive prototypes, two for Standard ML and one for Java, and used them for experiments to determine the practicality of our concepts.

In our experiments and proofs, we found that program metamorphosis tools are both more flexible than existing tools for interactive program evolution and give stronger behaviour-preservation guarantees, while offering practical interactive performance for nontrivial programs.

We thus argue that our results support our thesis statement and thereby suggest that program metamorphosis is a suitable replacement for traditional refactoring at least for programs up to moderate size.

Bibliography

[AB02] Zena M. Ariola and Stefan Blom. Skew confluence and the lambda calculus with letrec. Ann. Pure Appl. Logic, 117(1-3):95–168, 2002.

[ABB76] E. R. Anderson, F. C. Belz, and Edward K. Blum. SEMANOL (73) A Metalanguage for Programming the Semantics of Programming Languages. Acta Inf., 6:109–131, 1976.

[AK08] Malte Appeltauer and Günter Kniesel. Towards concrete syntax patterns for logic-based transformation rules. Electron. Notes Theor. Comput. Sci., 219:113–132, 2008.

[App92] Andrew W. Appel. Compiling with continuations. Cambridge University Press, New York, NY, USA, 1992.

[Ars79] Jacques J. Arsac. Syntactic source to source transforms and program manipulation. Commun. ACM, 22(1):43–54, 1979.

[Bac03] Roland C. Backhouse. Program Construction. Calculating Implementations from Specifications. John Wiley & Sons, Ltd., 2003.

[BBM$^+$85] F.L. Bauer, M. Broy, B. Möller, P. Pepper, M. Wirsing, et al. The Munich Project CIP. Vol. I: The Wide Spectrum Language CIP-L. Number 183 in Lecture Notes on Computer Science. Springer Verlag, Berlin, Heidelberg, New York, Berlin, 1985.

[BCD$^+$88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the System. In SDE 3: Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pages 14–24, New York, NY, USA, 1988. ACM Press.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. J. ACM, 24(1):44–67, 1977.

[Bec00] Kent Beck. eXtreme Programming eXplained, Embrace Change. Addison Wesley, 2000.

[BEH$^+$87] Friedrich L. Bauer, Herbert Ehler, A. Horsch, Bernhard Möller, Helmuth Partsch, O. Paukner, and Peter Pepper. The Munich Project CIP, Volume II: The Program Transformation System CIP-S, volume 292 of Lecture Notes in Computer Science. Springer, 1987.

[BG04] Marat Boshernitsan and Susan L. Graham. iXj: interactive source-to-source transformations for Java. j-SIGPLAN, 39(10):212–213, October 2004.

[BHM00] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. 2000.

[BKVV06] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.16: components for transformation systems. In PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pages 95–99, New York, NY, USA, 2006. ACM Press.

[BS86] Rolf Bahlke and Gregor Snelting. The psg system: from formal language definitions to interactive programming environments. ACM Trans. Program. Lang. Syst., 8(4):547–576, 1986.

[BS09] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In Proceedings of OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[BSCC04] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for object-oriented programming. Sci. Comput. Program., 52(1-3):53–100, 2004.

[CDKD86] Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language: mini-ML. In Proc. of the 1986 ACM conf. on LISP and Functional Prog., New York, NY, 1986.

[CDMS02] James R. Cordy, Thomas R. Dean, Andrew J. Malton, and Kevin A. Schneider. Source transformation in software engineering using the txl transformation system. Information & Software Technology, 44(13):827–837, 2002.

[CGM06] Tal Cohen, Joseph Gil, and Itay Maman. JTL – the Java Tools Language. In Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, 2006.

[Cha88] David R. Chase. Safety considerations for storage allocation optimizations. In PLDI, pages 1–10, 1988.

[Cor06] James R. Cordy. The TXL source transformation language, 2006.

[Cri95] Cristina Cornes and Judicael Courant and Jean-Christophe Filliâtre and Gérard Huet and Pascal Manoury and César Muñoz and Chetan Murthy and Catherine Parent et al. The Coq Proof Assistant - reference manual v 5.10, 1995.

[CTCC98] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object oriented programs. ACM Transactions on Software Engineering, 7(3), July 1998.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies. a tool for automatic formula manipulation with application to the church-rosser theorem. Indagationes Mathematicae, 34:381–392, 1972.

[DF94] R. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. ACM Transactions on Software Engineering, 3(2), April 1994.

[DJ05] Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[ES95] Gregor Engels and Andy Schürr. Encapsulated hierarchical graphs, graph types, and meta types. Electr. Notes Theor. Comput. Sci., 2, 1995.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993, volume 28(6), pages 237–247. ACM Press, New York, 1993.

[FW01] Matthew Fluet and Stephen Weeks. Contification using dominators. SIGPLAN Not., 36(10):2–13, 2001.

[Gar00] M. Garagnani. A Correct Algorithm for Efficient Planning with Preprocessed Domain Axioms, December 2000.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass., 2000.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. Proofs and Types. Cambridge University Press, 1989.

[GM06] Alejandra Garrido and José Mesguer. Formal specification and verification of java refactorings. In Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, Philadelphia, PA, USA, September 2006.

[GN90] W. G. Griswold and D. Notkin. Program restructuring as an aid to software maintenance. Technical report, University of Washington, Seattle, WA, USA, August 1990.

[Gog00] Joseph Goguen. Software Engineering with OBJ: Algebraic Specifications in Action. Kluwer, 2000.

[GP01] Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax with variable binding. Formal Aspects of Computing, 13:341–363, 2001.

[Gri92] William G. Griswold. Program restructuring as an aid to software maintenance. PhD thesis, Seattle, WA, USA, 1992.

[has97] Haskell 1.4, a non-strict, purely functional language. Technical report, Yale University, April 1997.

[HK99] Jan Heering and Paul Klint. Semantics of programming languages: a tool-oriented approach. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1999.

[HPM$^+$05] Pedro Rangel Henriques, Maria Joao Varanda Pereira, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic Generation of Language-based Tools Using the LISA System. IEE Proceedings, Software, 152(2):54–69, April 2005.

[HRD08] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Developing and Debugging Algebraic Specifications for Java classes. ACM Trans. Softw. Eng. Methodol., 17(3), 2008.

[Hug00] John Hughes. Generalising monads to arrows. Sci. Comput. Program., 37(1–3):67–111, 2000.

[JW93] 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, Charleston, South Carolina, pages 71–84, 1993.

[Kah] Stefan Kahrs. On the Static Analysis of Extended ML.

[Kah87] Gilles Kahn. Natural semantics. In STACS '87: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, pages 22–39, London, UK, 1987. Springer-Verlag.

[Kah99] Wolfram Kahl. The Term Graph Programming System HOPS. pages 136–149, March 1999.

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In ICSM, pages 736–743, 2001.

[Ker04] Joshua Kerievsky. Refactoring to Patterns (Addison-Wesley Signature Series). Addison-Wesley Professional, August 2004.

[KFK07] Adam Kiezun, Robert M. Fuhrer, and Markus Keller. Advanced Refactoring in Eclipse: Past, Present and Future. In First Workshop on Refactoring Tools, Berlin, 2007. https://netfiles.uiuc.edu/dig/RefactoringWorkshop/Presentations/AdvancedRefactoringInEclipse.pdf.

[KK04] Günter Kniesel and Helge Koch. Static composition of refactorings. Sci. Comput. Program., 52(1-3):9–51, 2004.

[KNG07] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In ICSE '07: Proceedings of the 29th international conference on Software Engineering, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[Knu68] Donald E. Knuth. Semantics of context-free languages. Theory of Computing Systems, 2(2):127–145, June 1968.

[KS98] Stefan Kahrs and Donald Sannella. Reflections on the design of a specification language. Lecture Notes in Computer Science, 1382:154+, 1998.

[KST97] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML: A gentle introduction. Theoretical Computer Science, 173(2):445–484, 1997.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 47–57, New York, NY, USA, 1988. ACM Press.

[LMC03] Sorin Lerner, Todd D. Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In PLDI, pages 220–231, 2003.

[LMC05] Sorin Lerner, Todd D. Millstein, and Craig Chambers. Cobalt: A language for writing provably-sound compiler optimizations. Electr. Notes Theor. Comput. Sci., 132(1):5–17, 2005.

[LMRC05] Sorin Lerner, Todd D. Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In POPL, pages 364–377, 2005.

[Lov77] David B. Loveman. Program Improvement by Source-to-Source Transformation. J. ACM, 24(1):121–145, 1977.

[LRT03] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell, pages 27–38, New York, NY, USA, 2003. ACM Press.

[LT05] Huiqing Li and Simon Thompson. Formalisation of Haskell Refactorings. In Marko van Eekelen and Kevin Hammond, editors, Trends in Functional Programming, September 2005.

[LTR05] Huiqing Li, Simon Thompson, and Claus Reinke. The Haskell Refactorer, HaRe, and its API. Electr. Notes Theor. Comput. Sci., 141(4):29–34, 2005.

[LV97] B. Luttik and E. Visser. Specification of rewriting strategies, 1997.

[MCC+01] Andrew Malton, James R. Cordy, Darren Cousineau, Kevin A. Schneider, Thomas R. Dean, and Jason Reynolds. Processing software source text in automated design recovery and transformation. In IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension, page 127, Washington, DC, USA, 2001. IEEE Computer Society.

[MDJ02] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In ICGT '02: Proceedings of the First International Conference on Graph Transformation, pages 286–301, London, UK, 2002. Springer-Verlag.

[MEDJ05] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations: Research articles. J. Softw. Maint. Evol., 17(4):247–276, 2005.

[MNB$^+$94] Lawrence Markosian, Philip Newcomb, Russell Brand, Scott Burson, and Ted Kitzmiller. Using an enabling technology to reengineer legacy systems. Commun. ACM, 37(5):58–70, 1994.

[Mog88] Eugenio Moggi. Computational lambda-calculus and monads. pages 14–23. IEEE Computer Society Press, 1988.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[MSHD07] Todd Mytkowicz, Peter F. Sweeney, Matthias Hauswirth, and Amer Diwan. Time interpolation: So many metrics, so few registers. In MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pages 286–300, Washington, DC, USA, 2007. IEEE Computer Society.

[MT04] Tom Mens and Tom Tourw'e. A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2):126–139, February 2004.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David McQueen. The Definition of Standard ML (Revised). MIT Press, 1997.

[NFF$^+$05] Alexander Nareyek, Eugene C. Freuder, Robert Fourer, Enrico Giunchiglia, Robert P. Goldman, Henry Kautz, Jussi Rintanen, and Austin Tate. Constraints and ai planning. IEEE Intelligent Systems, 20(2):62–72, 2005.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[Opd92] William F. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[Pau94] Lawrence C. Paulson. Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow), volume 828 of Lecture Notes in Computer Science. Springer, 1994.

[PF05] Adrian Pop and Peter Fritzson. Debugging natural semantics specifications. In AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging, pages 77–82, New York, NY, USA, 2005. ACM Press.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, Denmark, University of Aarhus, 1981.

[PR05] François Pottier and Didier Rémy. The Essence of ML Type Inference. In Benjamin C. Pierce, editor, Advanced Topics in Types and Programming Languages, chapter 10, pages 389–489. MIT Press, 2005.

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. Theor. Pract. Object Syst., 3(4):253–263, 1997.

[RG09] Travis Alexander Rupp-Greene. Analysis of software evolution over time. department of computer science, university of colorado at boulder, 2009.

[RN03] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[Ros96] Kenneth A. Ross. Tail recursion elimination in deductive databases. ACM Transactions on Database Systems, 21(2):208–237, 1996.

[RT84] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, pages 42–48, New York, NY, USA, 1984. ACM Press.

[SDF$^+$03] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. The Java Developers Guide to Eclipse. Addison-Wesley, May 2003.

[SEdM08a] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Challenge proposal: verification of refactorings. In PLPV '09: Proceedings of the 3rd workshop on Programming languages meets program verification, pages 67–72, New York, NY, USA, 2008. ACM.

[SEdM08b] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for java. In OOPSLA, pages 277–294, 2008.

[SHM09] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: a vm-centric approach. SIGPLAN Not., 44(6):1–12, 2009.

[Sof] IntelliJ Software. Intellij idea. http://www.intellij.com/idea/.

[SSD01] T. Schrijvers, A. Serebrenik, and B. Demoen. Refactoring prolog programs, 2001.

[ST09] Friedrich Steimann and Andreas Thies. From public to private to absent: Refactoring java programs under constrained accessibility. In Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming, pages 419–443, Berlin, Heidelberg, 2009. Springer-Verlag.

[SVEdM09] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon. pages 369–393. 2009.

[TDX07] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of API refactorings in libraries. In ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pages 377–380, New York, NY, USA, 2007. ACM.

[TJ91] Jean P. Talpin and Pierre Jouvelot. Polymorphic type region and effect inference. Technical Report EMP-CRI E/150, 1991.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.

[TR01a] S. Thompson and C. Reinke. Refactoring functional programs, 2001.

[TR01b] Simon Thompson and Claus Reinke. A Catalogue of Functional Refactorings, Version 1, 2001.

[Tur36] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. 42:230–265, 1936. This is the paper that introduced what is now called the Universal Turing Machine.

[Ull89] J. D. Ullman. Bottom-up beats top-down for datalog. In Proc. of the 8th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pages 140–149, New York, NY, 1989. ACM Press.

[vD97] A. van Deursen. A Comparison of Software Refinery and ASF+SDF, 1997.

[vdBvDH+01] Mark G. J. van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In CC '01: Proceedings of the 10th International Conference on Compiler Construction, pages 365–370, London, UK, 2001. Springer-Verlag.

[vDD08] Daniel von Dincklage and Amer Diwan. Explaining failures of program analyses. In PLDI, pages 260–269, 2008.

[Vis99] Eelco Visser. Strategic pattern matching. In RtA '99: Proceedings of the 10th International Conference on Rewriting Techniques and Applications, pages 30–44, London, UK, 1999. Springer-Verlag.

[Vis04] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. Lecture Notes in Computer Science, 3016:216–238, June 2004.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog and binary decision diagrams for program analysis. In Kwangkeun Yi, editor, Proc. of the 3rd Asian Symp. on Prog. Lang. and Systems, volume 3780 of Lecture Notes in Computer Science. Springer-Verlag, November 2005.

[Wad89] Philip Wadler. Theorems for free! In Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989, pages 347–359, New York, 1989. ACM Press.

[Wad90a] P. L. Wadler. Comprehending Monads. In Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, pages 61–78, New York, NY, 1990. ACM.

[Wad90b] Philip Wadler. Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci., 73(2):231–248, 1990.

[Wad92] Philip Wadler. The Essence of Functional Programming. In Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 1–14, Albequerque, New Mexico, 1992.

[WZ05] Martin Ward and Hussein Zedan. MetaWSL and Meta-Transformations in the FermaT Transformation System. In COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1, pages 233–238, Washington, DC, USA, 2005. IEEE Computer Society.

[YYC08] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. SIGPLAN Not., 43(1):221–234, 2008.

[ZCW$^+$02] W. Zhao, B. Cai, D. Whalley, M. Bailey, R. van Engelen, X. Yuan, J. Hiser, J. Davidson, K. Gallivan, and D. Jones. VISTA: A System for Interactive Code Improvement. In Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems, pages 155–164. ACM Press, 2002.

Appendix A

Scope ML

Syntactically, Scope ML comprises Standard ML **let** expressions, **val** and **fun** definitions, and identifier occurrences. **fun** definitions may contain parameters. As in Standard ML, **val** and **fun** definitions may define multiple identifiers simultaneously (connected via and): simultaneous fun definitions may be mutually recursive, while simultaneous val definitions cannot see each other in scope.

In the following, we refer to val and fun definitions (which may contain multiple definitions via and) as *definition blocks*.

Scope ML is sufficient to showcase many of the issues we deal with while being complex enough to allow us to discuss the most prominent errors arising as part of nominal and structural transformations. Moreover, the solutions we develop for Scope ML are either directly applicable or extend straightforwardly to the entirety of Standard ML. We choose Scope ML over other languages such as Mini-ML [CDKD86] since most "restricted ML" variants focus on the dynamic semantics of ML, whereas our needs are largely due to syntax and static semantics.

The following BNF grammar defines the syntax of Scope ML:

$$\begin{array}{rcl}
\langle \textit{program} \rangle & ::= & \langle \textit{decls} \rangle \\
\langle \textit{decls} \rangle & ::= & \varepsilon \mid \langle \textit{decl} \rangle\ \langle \textit{decls} \rangle \\
\langle \textit{decl} \rangle & ::= & \mathsf{val}\ \langle \textit{vbs} \rangle \mid \mathsf{fun}\ \langle \textit{fbs} \rangle \\
\langle \textit{vbs} \rangle & ::= & \langle \textit{vb} \rangle \mid \langle \textit{vb} \rangle\ \mathsf{and}\ \langle \textit{vbs} \rangle \\
\langle \textit{vb} \rangle & ::= & \langle \textit{name} \rangle\ =\ \langle \textit{expr} \rangle \\
\langle \textit{fbs} \rangle & ::= & \langle \textit{fb} \rangle \mid \langle \textit{fb} \rangle\ \mathsf{and}\ \langle \textit{fbs} \rangle \\
\langle \textit{fb} \rangle & ::= & \langle \textit{name} \rangle\ \langle \textit{name} \rangle^{+}\ =\ \langle \textit{expr} \rangle \\
\langle \textit{expr} \rangle & ::= & \langle \textit{name} \rangle \mid \langle \textit{expr} \rangle\ \langle \textit{expr} \rangle \\
 & \mid & \mathsf{let}\ \langle \textit{decls} \rangle\ \mathsf{in}\ \langle \textit{expr} \rangle\ \mathsf{end}
\end{array}$$

Scope ML includes value ("val") and function ("fun") declarations, with functions taking parameters. Both values and functions can be defined in parallel (using "and"), which means different things: "val" bindings, if done in parallel, prevent the bodies of their declarations from accessing the identifiers defined in the same "val" block; this allows e.g. val x = y and y = x to be used to swap the names of two values. "fun" bindings, on the other hand, are allowed to be mutually recursive when combined via "and", i.e. the bodies B_1, B_2 of fun f_1 = B_1 and fun f_2 = B_2 may both reference the identifiers of f_1 and f_2. This behaviour reflects the semantics of Standard ML.

In the following, we will refer to any ⟨*decl*⟩ production as a *declaration block*.

In the above grammar, ⟨*name*⟩ refers to legal names of identifiers. We assume that we have replaced such names *n* by unique identifiers *i* during semantic analysis (matching up identifier uses with definitions), and that we have stored appropriate facts of the form $i \overset{\mathsf{n}}{\mapsto} n$ as part of our initial program model.

Appendix B

Automatic fixup search

We evaluated the automatic fixup search from our initial Datalog prototype by applying it to three programs, a four-line sample program into which we had introduced a simple name capture and two programs from the MLton benchmark suite: "life", a game of life implementation (157 loc), and "ray", a ray tracer (459 loc). We did not run on larger programs due to an unresolved implementation bug in our frontend that yielded incomplete initial program models. For our tests, we resolved this limitation by manually adding the missing information.

To determine run-time performance, we implemented two systems, a Datalog implementation and a manually-tuned implementation. However, for some of our tests the Datalog implementation was slower by up to six orders of magnitude, rendering it impractical for recovery planning; as a consequence, we only report numbers for our manual implementation in Table B.1.

The time for initialising the desired program model and for inferring inconsistencies is insignificant for the programs we considered and appears to scale roughly linearly with program size.

Experiences with Recovery Plan Search

We implemented two planners: one heuristic planner, for our Datalog system, and one dumb planner, performing greedy search, for our ad-hoc system. While both systems are currently not sufficiently efficient for practical use, they report promising results for small examples.

Consider the following program:

```
val num = 23
val result = f(42)
val x = num + 1
val k = x + result
```

Program	**Init (best)**	**Inconsistency (best)**
sample	$42\mu s$	$22\mu s$
life	$182\mu s$	$59\mu s$
ray	$571\mu s$	$112\mu s$

Table B.1: Datalog prototype: Inconsistency search performance. 'Init' refers to the time needed to abstract the desired program model from ground facts. 'Inconsistency' is worst-case (whole-program) inconsistency inference.

If we now rename num to ‘result,’ the program becomes inconsistent, as we introduce name capture. In the above case, both of our planners find numerous useful plans, including

- Undo the renaming of num to ‘result’
- Rename the original result to a fresh name
- Relocate the definition of the former num (now ‘result’) into the same **val** definition as x (so that the body of the definition of x cannot see this definition)
- Relocate the definition of x into the block in which the initial num is defined, for the same effect.

Our Datalog planner suffers mostly from slow inconsistency inference. Our ad-hoc planner eliminates this problem, but does not effectively prune the search space. Combining the efficient search space pruning of our Datalog system with the fast evaluation times of our ad-hoc planner would be likely to yield a planner that is practical at least for small and medium-sized programs.

Appendix C

Congruence proof

With this type system, we obtain a useful result that our earlier definition of congruence-neutrality already relied on:

Lemma 1. *λ_{cc} is strongly normalising.*

Proof. Our type system is strictly less permissive than the type system for System F, which is strongly normalising [GLT89]. □

However, to do so we must require that none of our constants contain congruence casts or behave as if they do, and that our constants are monomorphic. To see why the latter is necessary, consider a function $\mathsf{id} : \forall\alpha.\alpha \to \alpha$. With this function, we can construct a term

$$\diamondminus^{-1}(\mathsf{id}(\diamondminus^{1}(1)))$$

that will be well-typed as $0 \boxplus \bullet$ and, for all intents and purposes, 'do the right thing' if we *know* what this library function does. However, we cannot reduce this term with our reduction rules, and thus we cannot eliminate the congruence casts in this case without exploiting parametricity [Wad89]; for brevity, we do not expand on this idea here.

Soundness proof.

Some well-typed terms, such as $\diamondminus^{1}(c) : 1 + \bullet$, contain a congruence cast, and we can discern this from their type. However, we can reduce any term that does not 'discernibly' contain a cast (looking at the type alone) into a normal form that contains no casts at all.

To show this, we must first define what it means for a term to 'discernibly' contain a congruence cast:

Definition 2. *We say that a type τ is neutral iff any $i \in \mathbb{Z}$ that syntactically occur in τ are $i = 0$, and overload this notion to say that a well-typed term $t : \tau$ is neutral iff τ is neutral.*

As we previously argued, none of our constants $c \in \mathit{Const}$ must 'contain a congruence cast.' We can now formalise this by requiring that all such c be neutral.

Next, we define our postulated congruence-cast-free normal form:

Definition 3. *We say that a term t in λ_{cc} is a value iff t does not contain coercion operators, and all applications syntactically occurring in t are of the form $@(z_1, z_2)$ with $z_1, z_2 \in \mathit{Var} \cup \mathit{Const}$.*

We note that any value is neutral, and any subterm of a value is also a value.

It is easy to see that none of our reduction rules applies to a value.

The only 'legal' way we have for reducing a term to a value at a normal form is to apply our reduction rules. Thus, before we can explain how we can reduce our terms in this fashion, we must show that our reduction rules will preserve our types:

Lemma 2. *(Subject Reduction.) For any term $t : \tau$, if $t \Rightarrow t'$, then $t' : \tau$.*

Proof. For our three reduction rules, ($\diamondminus_0$) and ($\diamondminus_1$) correspond directly to type equivalences. The only interesting case is (β_{cc}), but this proof is straightforward by structural induction over the function body (showing that term substitution locally preserves all types). □

Before we show our main lemma, we introduce a technical lemma that shows us that certain kinds of terms are always reducible:

Lemma 3. *Assume that all free variables are neutral. Let $t = @(t_1, t_2)$ well-typed and t_1 non-neutral. Then t is reducible.*

Proof. We show this by induction over the structure of subterms of t, excluding λ-bound variables: none of the subterms whose properties we require can be λ-bound variables.

First, note that if $t' = @(t'_1, t'_2)$ is non-neutral, then t'_1 must also be non-neutral.

t_1 cannot be a variable or constant, as those are neutral. If t_1 is a function abstraction, it is reducible. If t_1 is another function application, our argument holds by induction. The only interesting case is if $t_1 = \diamondminus^i(t')$. If $i = 0$, then this case is trivially reducible. Otherwise recall that (*app*) requires $\tau = 0 \boxplus \tau'$, so either $t' = \diamondminus^k(t'')$, making t reducible, or $t' = @(t'_1, t'_2)$, making t' reducible by induction hypothesis. □

Lemma 4. *(Progress.) Let t neutral. Assume that Z contains all free variables in t and all $z \in Z$ are neutral. Then either t is a value, or we can reduce t.*

Proof. Proof by induction over the inverse structure of t:

- we *assume* t is neutral if all elements of Z are
- we *show* that one of the following holds:
  - t or one of the subterms of t is reducible
  - If all of the subterms of t are values, then so is t.

The assumption for our induction hypothesis holds by the premise for this lemma. First, note that variables are trivial.

function abstraction. $t = \lambda x.t'$: With t neutral, hence so are x and t'. With $Z' = Z \cup \{x\}$ we have that if t' is irreducible, it must be a value, and then so is t.

function application. $t = @(t_1, t_2)$. If t_1 is not neutral, then t is reducible by Lemma 3. Otherwise both t_1 and t_2 are neutral (and, hence, either reducible or values). The only interesting case then is if t_1 and t_2 are both values, and $t_1 = (\lambda x.t')$, in which case t is reducible.

congruence cast. $t = \diamondminus^i(t')$. Either $i = 0$, in which case this rule is reducible via ($\diamondminus_0$), or $t' : -i \boxplus \tau$. To have this type, either $t' = \diamondminus^k(t'')$, which is reducible via ($\diamondminus_1$), or $t' = @(t'_1, t'_2)$, which is reducible by Lemma 3. □

Our main result from this section is then the following:

Theorem 1. *Let t neutral, and assume that all $c \in Const$ neutral and for all such $c : \tau$, τ contains no α. Then t is congruence-neutral.*

Proof. By lemmata 4 and 2 t reduces to a value. □

Appendix D

Soundness proof for the Simple ML program models

We now show the soundness of our program models from Section 6.2.3.

We define the Simple ML semantics as a relation between *program state*s. Program state is a current expression together with an *interpreter state*, $\mathcal{S} = \langle \phi, \sigma \rangle$, that captures program output as $\phi \in Val^*$ and a store as $\sigma \in Addr \to Val$.

We use the notation $\mathcal{S}.\phi$ and $\mathcal{S}.\sigma$ to project the first and second component, respectively, and further define update notation, $\mathcal{S}[\sigma \to f(\sigma)]$ that allows us to update components of our state. For example, $\mathcal{S}[\sigma \to \sigma\{a \mapsto \mathsf{true}\}]$ would update the store component of $\mathcal{S}$ to map a to true.

More concisely, our update notation denotes

$$\mathcal{S}[\sigma \to f(\sigma)] \overset{\Delta}{=} \mathsf{let}\ \langle \phi, \sigma \rangle = \mathcal{S}\ \mathsf{in}\ \langle \phi, f(\sigma) \rangle$$

and analogously for ϕ.

Definition 4. *The simple ML interpreter is the minimal binary relation ($\Downarrow$) defined in Figure D.1. This interpreter relates tuples of program state $\langle e, \mathcal{S} \rangle$, where e is an expression and $\mathcal{S} = \langle \phi, \sigma \rangle$.*

Note that Simple ML is confluent.

We require the toplevel expression in our programs to be a function application with a 'pure' (effect-free) parameter. This limitation is purely syntactic and simplifies our later proofs:

Definition 5. *A Simple ML program $e \in Expr$ is any production of the form $(\lambda x.e)$**()** as defined in Figure 6.1 that is closed with respect to Var, i.e., does not contain free variables.*

If we assume a standard ML-style typing regime (omitted for brevity), our interpreter yields an irreducible value, with the set of values as follows:

Definition 6. *A Simple ML value $\overline{e} \in Val$ is either an address $a \in Addr$, a lambda abstraction, a constant, or an exception (signified by* raise*).*

We now show that our two models together guarantee behaviour preservation modulo nontermination. We first define:

Definition 7. *The program model of a program $p : t$ is $\langle t, \mathfrak{E}_p \rangle$.*

To show soundness, we then first show that if we compute a program model M_p for a program p, then M_p prescribes precisely the result of evaluating p. Then we show that any M'_p that is equivalent to M_p prescribes exactly the same evaluation result modulo renaming and nontermination.

environment

$$\frac{}{E \vdash \langle x, \mathcal{S} \rangle \Downarrow \langle E(x), \mathcal{S} \rangle} \; (\mathit{var})$$

beta reduction

$$\frac{\begin{array}{c} E \vdash \langle e_1, \mathcal{S} \rangle \Downarrow \langle \lambda x.e_3, \mathcal{S}' \rangle \\ E \vdash \langle e_2, \mathcal{S}' \rangle \Downarrow \langle \overline{e_2'}, \mathcal{S}'' \rangle \\ E\{x \mapsto e_2'\} \vdash \langle e_3, \mathcal{S}'' \rangle \Downarrow \langle \overline{e_3'}, \mathcal{S}''' \rangle \end{array}}{E \vdash \langle e_1\ e_2, \mathcal{S} \rangle \Downarrow \langle e_3', \mathcal{S}''' \rangle} \; (\beta)$$

$$\frac{E \vdash \langle e_1, \mathcal{S} \rangle \Downarrow \langle \textsf{raise}, \mathcal{S}' \rangle}{E \vdash \langle e_1\ e_2, \mathcal{S} \rangle \Downarrow \langle \textsf{raise}, \mathcal{S}' \rangle} \; (\beta\text{-}x\text{-}1)$$

$$\frac{\begin{array}{c} E \vdash \langle e_1, \mathcal{S} \rangle \Downarrow \langle \lambda x.e_3, \mathcal{S}' \rangle \\ E \vdash \langle e_2, \mathcal{S}' \rangle \Downarrow \langle \textsf{raise}, \mathcal{S}'' \rangle \end{array}}{E \vdash \langle e_1\ e_2, \mathcal{S} \rangle \Downarrow \langle \textsf{raise}, \mathcal{S}'' \rangle} \; (\beta\text{-}x\text{-}2)$$

conditionals

$$\frac{E \vdash \langle e_1, \mathcal{S} \rangle \Downarrow \langle \textsf{true}, \mathcal{S}' \rangle \quad E \vdash \langle e_2, \mathcal{S} \rangle \Downarrow \langle \overline{e_2'}, \mathcal{S}' \rangle}{E \vdash \langle \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3, \mathcal{S} \rangle \Downarrow \langle e_2', \mathcal{S}'' \rangle} \; (\mathit{if\text{-}t})$$

$$\frac{E \vdash \langle e_1, \mathcal{S} \rangle \Downarrow \langle \textsf{false}, \mathcal{S}' \rangle \quad E \vdash \langle e_3, \mathcal{S} \rangle \Downarrow \langle \overline{e_3'}, \mathcal{S}' \rangle}{E \vdash \langle \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3, \mathcal{S} \rangle \Downarrow \langle e_3', \mathcal{S}'' \rangle} \; (\mathit{if\text{-}f})$$

$$\frac{E \vdash \langle e_1, \mathcal{S} \rangle \Downarrow \langle \textsf{raise}, \mathcal{S}' \rangle}{E \vdash \langle \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3, \mathcal{S} \rangle \Downarrow \langle \textsf{raise}, \mathcal{S}' \rangle} \; (\mathit{if\text{-}x})$$

exception handling

$$\frac{E \vdash \langle e_1, \mathcal{S} \rangle \Downarrow \langle \overline{e_1'}, \mathcal{S}' \rangle}{E \vdash \langle e_1 \textsf{ handle } e_2, \mathcal{S} \rangle \Downarrow \langle e_1', \mathcal{S}' \rangle} \; (\mathit{handle\text{-}0})$$

$$\frac{E \vdash \langle e_1, \mathcal{S} \rangle \Downarrow \langle \textsf{raise}, \mathcal{S}' \rangle \quad E \vdash \langle e_2, \mathcal{S}' \rangle \Downarrow \langle \overline{e_2'}, \mathcal{S}'' \rangle}{E \vdash \langle e_1 \textsf{ handle } e_2, \mathcal{S} \rangle \Downarrow \langle e_2', \mathcal{S}'' \rangle} \; (\mathit{handle})$$

Figure D.1: Simple ML semantics: functions, conditionals, exceptions.

let

$$\frac{E \vdash \langle e_1, \mathcal{S}\rangle \Downarrow \langle \overline{e'_1}, \mathcal{S}'\rangle \quad E\{x \mapsto e'_1\} \vdash \langle e_2, \mathcal{S}'\rangle \Downarrow \langle \overline{e'_2}, \mathcal{S}''\rangle}{E \vdash \langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \mathcal{S}\rangle \Downarrow \langle e'_2, \mathcal{S}''\rangle}\ (\mathit{let})$$

- -

$$\frac{E \vdash \langle e_1, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}'\rangle}{E \vdash \langle \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}'\rangle}\ (\mathit{let\text{-}x})$$

printing

$$\frac{E \vdash \langle e_1, \mathcal{S}\rangle \Downarrow \langle \overline{e'_1}, \mathcal{S}'\rangle}{E \vdash \langle \mathsf{print}\ e_1, \mathcal{S}\rangle \Downarrow \langle \mathbf{()}, \mathcal{S}'[\phi \to e'_1 :: \phi]\rangle}\ (\mathit{print})$$

- -

$$\frac{E \vdash \langle e, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}'\rangle}{E \vdash \langle \mathsf{print}\ e, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}'\rangle}\ (\mathit{print\text{-}x})$$

ref values

$$\frac{E \vdash \langle e, \mathcal{S}\rangle \Downarrow \langle \overline{e'}, \mathcal{S}'\rangle \quad a\ \mathit{fresh}}{E \vdash \langle \mathsf{ref}\ e, \mathcal{S}\rangle \Downarrow \langle a, \mathcal{S}'[\sigma \to \sigma\{a \mapsto e'\}]\rangle}\ (\mathit{ref})$$

$$\frac{E \vdash \langle e, \mathcal{S}\rangle \Downarrow \langle a, \mathcal{S}'\rangle}{E \vdash \langle !e, \mathcal{S}\rangle \Downarrow \langle \mathcal{S}'.\sigma(a), \mathcal{S}'\rangle}\ (\mathit{read})$$

$$\frac{\begin{array}{c} E \vdash \langle e_1, \mathcal{S}\rangle \Downarrow \langle a, \mathcal{S}'\rangle \\ E \vdash \langle e_2, \mathcal{S}'\rangle \Downarrow \langle \overline{e'_2}, \mathcal{S}''\rangle \end{array}}{E \vdash \langle e_1{:=}e_2, \mathcal{S}\rangle \Downarrow \langle \mathbf{()}, \mathcal{S}''[\sigma \to \sigma\{a \mapsto e'_2\}]\rangle}\ (a)$$

- -

$$\frac{E \vdash \langle e, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}'\rangle}{E \vdash \langle \mathsf{ref}\ e, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}'\rangle}\ (\mathit{ref\text{-}x})$$

$$\frac{E \vdash \langle e, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}'\rangle}{E \vdash \langle !e, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}'\rangle}\ (\mathit{read\text{-}x})$$

$$\frac{E \vdash \langle e_1, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}'\rangle}{E \vdash \langle e_1{:=}e_2, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}'\rangle}\ (\mathit{a\text{-}x\text{-}1})$$

$$\frac{\begin{array}{c} E \vdash \langle e_1, \mathcal{S}\rangle \Downarrow \langle \overline{e'_1}, \mathcal{S}'\rangle \\ E \vdash \langle e_2, \mathcal{S}'\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}''\rangle \end{array}}{E \vdash \langle e_1{:=}e_2, \mathcal{S}\rangle \Downarrow \langle \mathsf{raise}, \mathcal{S}''\rangle}\ (\mathit{a\text{-}x\text{-}2})$$

Figure D.1: Simple ML semantics: let, prints, and assignments.

We now construct a *model interpreter* that 'executes' program model terms and effect sequences, and show that the model interpreter yields identical evaluation results identical to what our Simple ML operational semantics yield on the source program.

For side effects, we use a single effect sequence that we thread through our interpreter in a monadic fashion.

To show that program models prescribe the evaluation result, we divide the notion of the 'evaluation result' up into three parts:

- The exception state, i.e., whether the evaluation yields a result or the 'quasi-value' raise.
- The computation result, unless the evaluation yields raise.
- The term evaluation state, denoted $\mathcal{T}$, where $\mathcal{T} = \langle \varsigma, \varphi, \varrho \rangle$ in turn consists of three components that we project and update with notation analogous to Definition 4:
    - $\varsigma \in$ *TermAddr* $\rightarrow$ *Term*: the *term store* maps addresses to terms.
    - $\varphi \in \varphi \in$ *Term**: the *output* is a list of expressions.
    - $\varrho \in$ *RIndex* $\rightarrow$ *Addr*: the *read store* maps read IDs to addresses. This allows us to decouple read effects (`r`), which are subject to partial ordering with respect to other effects, from read terms (**read**), which are not.

Definition 8. *The model interpreter is the minimal relation $\downarrow$ defined in Figure D.2. The evaluation relation $\downarrow$ relates model state tuples $\langle t, \eta, \mathcal{T} \rangle$, where we call t the term, η the effect trace, and $\mathcal{T}$ the state.*

Analogously to our definition of expressions, we define that a term t is a value term, notation $\bar{t}$, iff there are no $\eta, \mathcal{T}, t', \mathcal{T}', \eta'$ such that $\langle t, \eta, \mathcal{T} \rangle \downarrow \langle t', \eta', \mathcal{T}' \rangle$.

Definition 9. *For effect traces, we say that a trace η is terminal with respect to a term t iff either $\eta = \bigcirc$ or otherwise both $\eta = \mathsf{exn}_\blacklozenge$ and $t = \textbf{raise}_\blacklozenge$.*

Definition 10. *With relation to Definition 4, we say that a model state $\langle t, \eta, \mathcal{T} \rangle$ models a program state $\langle e, \mathcal{S} \rangle$, notation $\langle e, \mathcal{S} \rangle \simeq \langle e, \eta, \mathcal{T} \rangle$, iff all of the following hold:*

(1) $e : t$

(2) *Let* $\mathcal{S}.\phi = [v_1, \ldots, v_n]$. *Then* $\mathcal{T}.\varphi = [t_1, \ldots, t_n]$, *with* $v_i : t_i$ *for all* $i \in \{1, \ldots, n\}$.

(3) *dom* $\mathcal{S}.\sigma =$*dom* $\mathcal{T}.\varsigma$, *and for all* $a \in$*dom* $\mathcal{S}.\sigma, \mathcal{S}.\sigma(a) : \mathcal{T}\varsigma(a)$

We overload this operator for environments, with $E \simeq \Sigma$ iff dom $E =$dom Σ and for all $x \in$dom E, $E(x) : \Sigma(x)$.

Note that exceptions are evaluated iff they are present both in the term and on top of the effect trace. We can then see the following:

Corollary 1. $\bar{e} : \bar{t}$*, i.e., a term that describes a value is a value term.*

let and beta reduction

$$\frac{\begin{array}{c}\Sigma \vdash \langle t_1, \eta, \mathcal{T}\rangle \downarrow \langle \lambda_k x.t'_1, \eta', \mathcal{T}'\rangle \\ \Sigma \vdash \langle t_2, \eta', \mathcal{T}'\rangle \downarrow \langle \overline{t'_2}, @_\theta(t_1, t_2) : \eta_f \triangleright \eta'', \mathcal{T}''\rangle \\ \Sigma\{x \mapsto t'_2\} \vdash \langle t'_1, \mathfrak{E}(k), \mathcal{T}''[\varrho \to \{\}]\rangle \downarrow \langle \overline{t'}, \bigcirc, \mathcal{T}'''\rangle\end{array}}{\Sigma \vdash \langle @_\theta(t_1, t_2), \eta, \mathcal{T}\rangle \downarrow \langle t', \eta'', \mathcal{T}'''[\varrho \to \mathcal{T}''.\varrho]\rangle}\ (M\text{-}\beta)$$

$$\frac{\begin{array}{c}\Sigma \vdash \langle t_1, \eta, \mathcal{T}\rangle \downarrow \langle \overline{t'_1}, \eta', \mathcal{T}'\rangle \\ \Sigma\{x \mapsto t'_1\} \vdash \langle t_2, \eta', \mathcal{T}'\rangle \downarrow \langle \overline{t'_2}, \mathcal{T}''\rangle\end{array}}{\Sigma \vdash \langle \textbf{let}\ x = t_1\ \textbf{in}\ t_2, \eta, \mathcal{T}\rangle \downarrow \langle t'_2, \eta'', \mathcal{T}''\rangle}\ (M\text{-}let)$$

- -

$$\frac{\begin{array}{c}\Sigma \vdash \langle t_1, \eta, \mathcal{T}\rangle \downarrow \langle \lambda_k x.t'_1, \eta', \mathcal{T}'\rangle \\ \Sigma \vdash \langle t_2, \eta', \mathcal{T}'\rangle \downarrow \langle \overline{t'_2}, @_\theta(t_1, t_2) : \eta_f \triangleright \eta'', \mathcal{T}''\rangle \\ \Sigma\{x \mapsto t'_2\} \vdash \langle t'_1, \mathfrak{E}(k), \mathcal{T}''[\varrho \to \{\}]\rangle \downarrow \langle \textbf{raise}_\blacklozenge, \mathrm{exn}_\blacklozenge, \mathcal{T}'''\rangle\end{array}}{\Sigma \vdash \langle @_\theta(t_1, t_2), \eta, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta'', \mathcal{T}'''[\varrho \to \mathcal{T}''.\varrho]\rangle}\ (M\text{-}\beta\text{-}x)$$

$$\frac{\Sigma \vdash \langle t_1, \eta, \varphi, \varsigma, \varrho\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta', \mathcal{T}'\rangle}{\Sigma \vdash \langle @_\theta(t_1, t_2), \eta, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta', \mathcal{T}'\rangle}\ (M\text{-}\beta\text{-}x\text{-}1)$$

$$\frac{\begin{array}{c}\Sigma \vdash \langle t_1, \eta, \mathcal{T}\rangle \downarrow \langle \lambda_k x.t'_1, \eta', \mathcal{T}'\rangle \\ \Sigma \vdash \langle t_2, \eta', \mathcal{T}'\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta'', \mathcal{T}''\rangle\end{array}}{\Sigma \vdash \langle @_\theta(t_1, t_2), \eta, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta'', \mathcal{T}''\rangle}\ (M\text{-}\beta\text{-}x\text{-}2)$$

$$\frac{\Sigma \vdash \langle t_1, \eta, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta', \mathcal{T}'\rangle}{\Sigma \vdash \langle \textbf{let}\ x = t_1\ \textbf{in}\ t_2, \eta, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta', \mathcal{T}''\rangle}\ (M\text{-}let\text{-}x)$$

conditionals

$$\frac{\Sigma \vdash \langle t_1, \eta, \mathcal{T}\rangle \downarrow \langle \mathsf{true}, \mathtt{ite}(t_1, \eta_2, \eta_3) \triangleright \eta_t, \mathcal{T}'\rangle \quad \Sigma \vdash \langle t_2, \eta_2, \mathcal{T}'\rangle \downarrow \langle \overline{t'_2}, \bigcirc, \mathcal{T}''\rangle}{\Sigma \vdash \langle \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3, \eta, \mathcal{T}\rangle \downarrow \langle t'_2, \eta_r \triangleright \eta_t, \mathcal{T}''\rangle}\ (M\text{-}if\text{-}t)$$

$$\frac{\Sigma \vdash \langle t_1, \eta_h, \mathcal{T}\rangle \downarrow \langle \mathsf{false}, \mathtt{ite}(t_1, \eta_2, \eta_3) \triangleright \eta_t, \mathcal{T}'\rangle \quad \Sigma \vdash \langle t_3, \eta_3, \mathcal{T}'\rangle \downarrow \langle \overline{t'_3}, \bigcirc, \mathcal{T}''\rangle}{\Sigma \vdash \langle \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3, \eta, \mathcal{T}\rangle \downarrow \langle t'_3, \eta_r \triangleright \eta_t, \mathcal{T}''\rangle}\ (M\text{-}if\text{-}f)$$

- -

$$\frac{\Sigma \vdash \langle t_1, \eta_h, \mathcal{T}\rangle \downarrow \langle \mathsf{raise}_\theta, \mathtt{exn}_\theta \triangleright \eta_t, \mathcal{T}'\rangle}{\Sigma \vdash \langle \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3, \eta, \mathcal{T}\rangle \downarrow \langle \mathsf{raise}_\theta, \mathtt{exn}_\theta \triangleright \eta_t, \mathcal{T}'\rangle}\ (M\text{-}if\text{-}x)$$

exception handling

$$\frac{\Sigma \vdash \langle t_1, \eta, \mathcal{T}\rangle \downarrow \langle \overline{t'_1}, \mathtt{handle}_\theta\ \eta_2 \triangleright \eta_t, \mathcal{T}'\rangle \quad t'_1 \neq \mathsf{raise}_{\theta'}}{\Sigma \vdash \langle t_1\ \mathsf{handle}_\theta\ t_2, \eta, \mathcal{T}\rangle \downarrow \langle t'_1, \eta_t, \mathcal{T}'\rangle}\ (M\text{-}handle\text{-}0)$$

- -

$$\frac{\begin{array}{c}\Sigma \vdash \langle t_1, \eta, \mathcal{T}\rangle \downarrow \langle \mathsf{raise}_\theta \triangleright \mathtt{handle}_\theta\ \eta_2 \triangleright \eta_t, \mathtt{exn}_\theta, \mathcal{T}'\rangle \\ \Sigma \vdash \langle t_2, \eta_2, \mathcal{T}'\rangle \downarrow \langle \overline{t'_2}, \overline{\eta'_2}, \mathcal{T}''\rangle\end{array}}{\Sigma \vdash \langle t_1\ \mathsf{handle}_\theta\ t_2, \eta, \mathcal{T}\rangle \downarrow \langle t'_2, \eta'_2 \triangleright \eta_t, \mathcal{T}'\rangle}\ (M\text{-}handle)$$

Figure D.2: The model interpreter: functions, conditionals, and exceptions.

environment

$$\frac{}{\Sigma \vdash \langle x, \eta, \mathcal{T}\rangle \downarrow \langle \Sigma(x), \eta, \mathcal{T}\rangle} \quad (\textit{M-var})$$

printing

$$\frac{\Sigma \vdash \langle t_p, \eta_h, \mathcal{T}\rangle \downarrow \langle \overline{t'_p}, \circ, \mathcal{T}'\rangle}{\Sigma \vdash \langle t, \eta_h \triangleright t_p : \texttt{io} \triangleright \eta_t, \mathcal{T}\rangle \downarrow \langle t, \eta_t, \mathcal{T}'[\varphi \to t'_p :: \varphi]\rangle} \quad (\textit{M-print})$$

$$\frac{\Sigma \vdash \langle t_p, \eta_h, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta, \mathcal{T}'\rangle}{\Sigma \vdash \langle t, \eta_h \triangleright t_p : \texttt{io} \triangleright \eta_t, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta_t, \mathcal{T}'\rangle} \quad (\textit{M-print-x})$$

ref values

$$\frac{\Sigma \vdash \langle t, \eta, \mathcal{T}\rangle \downarrow \langle \overline{t'}, t : \mathsf{a}_i \triangleright \eta', \mathcal{T}'\rangle \quad a \text{ fresh}}{\Sigma \vdash \langle \textbf{ref}_i, \eta, \mathcal{T}\rangle \downarrow \langle a, \eta', \mathcal{T}'[\varsigma \to \varsigma\{a \mapsto t'\}]\rangle} \quad (\textit{M-ref})$$

$$\frac{}{\Sigma \vdash \langle \textbf{read}_\rho, \eta, \mathcal{T}\rangle \downarrow \langle \mathcal{T}.\varrho(\rho), \eta, \mathcal{T}\rangle} \quad (\textit{M-read})$$

$$\frac{\Sigma \vdash \langle t_r, \eta_h, \mathcal{T}\rangle \downarrow \langle a, \circ, \mathcal{T}'\rangle}{\Sigma \vdash \langle t, \eta_h \triangleright t_r : \texttt{r}_\rho \triangleright \eta_t, \mathcal{T}\rangle \downarrow \langle t, \eta_t, \mathcal{T}'[\varrho \to \varrho\{\rho \mapsto \varsigma'(a)\}]\rangle} \quad (\textit{M-e-read})$$

$$\frac{\begin{array}{c}\Sigma \vdash \langle t_1, \eta_h, \mathcal{T}\rangle \downarrow \langle a, \eta'_h, \mathcal{T}'\rangle \\ \Sigma \vdash \langle t_2, \eta'_h, \mathcal{T}'\rangle \downarrow \langle \overline{t'_2}, \circ, \mathcal{T}''\rangle\end{array}}{\Sigma \vdash \langle t, \eta, \eta_h \triangleright t_1 \texttt{:=} t_2 \triangleright \eta_t, \mathcal{T}\rangle \downarrow \langle t, \eta_t, \mathcal{T}''[\varsigma \to \varsigma\{a \mapsto t'_2\}]\rangle} \quad (\textit{M-a})$$

$$\frac{\Sigma \vdash \langle t, \eta, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta', \mathcal{T}'\rangle}{\Sigma \vdash \langle \mathsf{ref}_i\ t, \mathsf{a}_i \triangleright \eta, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta', \mathcal{T}'\rangle} \quad (\textit{M-ref-x})$$

$$\frac{\Sigma \vdash \langle t_r, \eta_h, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta, \mathcal{T}'\rangle}{\Sigma \vdash \langle t, \eta_h \triangleright t_r : \texttt{r}_i \triangleright \eta_t, \mathcal{T}\rangle \downarrow \langle \textbf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta_t, \mathcal{T}'\rangle} \quad (\textit{M-read-x})$$

$$\frac{\Sigma \vdash \langle t_1, \eta_h, \mathcal{T}\rangle \downarrow \langle \mathsf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta'_h, \mathcal{T}'\rangle}{\Sigma \vdash \langle t, \eta, \eta_h \triangleright t_1 \texttt{:=} t_2 \triangleright \eta_t, \mathcal{T}\rangle \downarrow \langle \mathsf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta''_h \triangleright \eta_t, \mathcal{T}'\rangle} \quad (\textit{M-a-x-0})$$

$$\frac{\begin{array}{c}\Sigma \vdash \langle t_1, \eta_h, \varphi, \varsigma\rangle \downarrow \langle a, \eta'_h, \mathcal{T}'\rangle \\ \Sigma \vdash \langle t_2, \eta'_h, \varphi', \varsigma'\rangle \downarrow \langle \mathsf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta''_h, \mathcal{T}''\rangle\end{array}}{\Sigma \vdash \langle t, \eta, \eta_h \triangleright t_1 \texttt{:=} t_2 \triangleright \eta_t, \mathcal{T}\rangle \downarrow \langle \mathsf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta''_h \triangleright \eta_t, \mathcal{T}''\rangle} \quad (\textit{M-a-x-1})$$

effect management

$$\frac{\eta_h \neq \eta_1 \triangleright \eta_2 \quad \eta_h \neq \mathsf{handle}_{\theta'}}{\Sigma \vdash \langle \mathsf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta_h \triangleright \eta_t, \mathcal{T}\rangle \downarrow \langle \mathsf{raise}_\theta, \mathrm{exn}_\theta \triangleright \eta_t, \mathcal{T}\rangle} \quad (\textit{M-x-delete})$$

$$\frac{}{\Sigma \vdash \langle t, \circ \triangleright \eta, \mathcal{T}\rangle \downarrow \langle t, \eta, \mathcal{T}\rangle} \quad (\textit{M-empty})$$

Figure D.2: The model interpreter: let, prints, and assignments.

To give a flavour of the rules in our term interpreter, we consider some of them below before considering them in their entirety:

$$\frac{\Sigma \vdash \langle t_p, \eta_h, \mathcal{T}\rangle \downarrow \langle \overline{t_p'}, \bigcirc, \mathcal{T}'\rangle}{\Sigma \vdash \langle t, \eta_h \triangleright t_p : \mathtt{io} \triangleright \eta_t, \mathcal{T}\rangle \downarrow \langle t, \eta_t, \mathcal{T}'[\varphi \to t_p' :: \varphi]\rangle}\ (\textit{M-print})$$

This rule interprets print operations. Note that in the rule's conclusion, we have the same term t 'afterwards' (on the right side of the arrow) as we did 'before', though we have eliminated effects from the effect trace. This matches our interpretation of prints in models: any printing is only manifest in the effect model and is thus, in effect, asynchronous with respect to the main program's evaluation.

The rule's premise reduces the value-to-print, t_p, to an irreducible $\overline{t_p'}$ and ensures that all effects that must take place prior to the printing have taken place (by reducing η_h to $\bigcirc$). Then, it updates the 'print output' aspect, φ, of our term state $\mathcal{T}'$ by adding the most recently printed output.

Next, consider reading.

$$\frac{\Sigma \vdash \langle t_r, \eta_h, \mathcal{T}\rangle \downarrow \langle a, \bigcirc, \mathcal{T}'\rangle}{\Sigma \vdash \langle t, \eta_h \triangleright t_r : \mathtt{r}_\rho \triangleright \eta_t, \mathcal{T}\rangle \downarrow \langle t, \eta_t, \mathcal{T}'[\varrho \to \varrho\{\rho \mapsto \varsigma'(a)\}]\rangle}$$

As with printing, we execute the read independently of any term-we-wish-to-evaluate and store the result in our 'read store,' ϱ. If we then wish to access the read in a term, we use the following rule, (*M-e-read*):

$$\frac{}{\Sigma \vdash \langle \textbf{read}_\rho, \eta, \mathcal{T}\rangle \downarrow \langle \mathcal{T}.\varrho(\rho), \eta, \mathcal{T}\rangle}\ (\textit{M-read})$$

which evaluates a term $\textbf{read}_\rho$ to whichever value we map ρ to in our read store. It is important to note that we cannot use this rule if ρ is not yet mapped to a value: thereby, read indices act as a (relaxed) method of synchronisation between effects and terms.

Finally, consider beta reduction:

$$\frac{\begin{array}{c}\Sigma \vdash \langle t_1, \eta, \mathcal{T}\rangle \downarrow \langle \lambda_k x.t_1', \eta', \mathcal{T}'\rangle \\ \Sigma \vdash \langle t_2, \eta', \mathcal{T}'\rangle \downarrow \langle \overline{t_2'}, @_\theta(t_1, t_2) : \eta_f \triangleright \eta'', \mathcal{T}''\rangle \\ \Sigma\{x \mapsto t_2'\} \vdash \langle t_1', \mathfrak{E}(k), \mathcal{T}''[\varrho \to \{\}]\rangle \downarrow \langle \overline{t'}, \bigcirc, \mathcal{T}'''\rangle\end{array}}{\Sigma \vdash \langle @_\theta(t_1, t_2), \eta, \mathcal{T}\rangle \downarrow \langle t', \eta'', \mathcal{T}'''[\varrho \to \mathcal{T}''.\varrho]\rangle}\ (\textit{M-}\beta)$$

As we see in the conclusion, this rule applies for a function application $@_\theta(t_1, t_2)$. We now consider the premises in descending order: in the first premise, we evaluate t_1 to a function $\lambda_k x.t_1'$, reducing the effect trace η to η' as needed (for example, we might have to execute $\mathtt{r}$ effects to be able to evaluate a **read** term). Similarly, we update the term state to $\mathcal{T}$ as needed.

In the second premise, we evaluate the function application parameter, t_2, to the irreducible t_2'. Note that we require that the effect trace now has the form $@_\theta(t_1, t_2) : \eta_f \triangleright \eta''$, meaning that our function application *must* be next in line for execution: this enforces, for example, that any residual prints are executed before the function application, ensuring that we produce all output in order.

In the third premise, we update our term environment Σ to map variable x to t_2' and execute the function body t_1' together with its accompanying effect trace $\mathfrak{E}(k)$, indexed by lambda key k, until the term is irreducible and no effects are left (we handle exceptional cases separately). Note that in this final premise we start with an empty read environment, and we restore our old read environment in the final computation result in the conclusion. This step is critical, since missing entries in the read environment ‘force’ assignments that happen prior to reads (as we suggested when we introduced (*M-e-read*) and (*M-read*)). If we were not to clear the environment, we could, in a recursive function call, allow (*M-read*) to take place before its accompanying (*M-e-read*), thereby possibly yielding the wrong result.

As we have seen, our ruleset introduces a considerable amount of nondeterminism: specifically, (*M-a*) and (*M-print*) do not interact with the term, whereas (*M-β-pure*) and (*M-var*) do not interact with the effect trace. While this nondeterminism expresses the natural separation between truly *side*-effects and computational results, it complicates our reasoning. To work around this nondeterminism, we first show that our model interpreter *can* produce the ‘same’ result and the ‘same’ side effects as the Simple ML interpreter (Lemma 5) before showing that it is confluent (Lemma 7) and thereby *must* produce the same result. Finally, in Theorem 3 we show that the changes we permit to terms and effects (Figure 6.7 and Figure 6.7) do not affect the computational result.

Lemma 5. *(Simulation) Assume an expression that is a function evaluation* $(\lambda x.e_1)\overline{e_2}$ *with an irreducible argument, and*

- $\langle(\lambda x.e_1)\overline{e_2}, \mathcal{S}\rangle \simeq \langle t, \eta, \mathcal{T}\rangle$
- $E \vdash \langle(\lambda x.e_1)\overline{e_2}, \mathcal{S}\rangle \Downarrow \langle e', \mathcal{S}'\rangle$
- $E \simeq \Sigma$

then there exist $t', \mathcal{S}'$ *such that* $\Sigma \vdash \langle t_1\ t_2, \eta, \mathcal{T}\rangle \downarrow \langle t', \eta', \mathcal{T}'\rangle$ *and* $\langle e', \mathcal{S}\rangle \simeq \langle t', \eta', \mathcal{T}'\rangle$, *and* η' *is terminal with respect to* t'.

Proof. By induction over the derivation via ($\Downarrow$). In the following, we use ‘expression’ to denote an expression in Simple ML and ‘term’ to denote a term as generated by our term model generation.

First, observe that we require e_2 irreducible. Thus, the next derivation step for ($\Downarrow$) must be (β). Syntactically, the next derivation step for ($\downarrow$) must be one of (*M-β*), (*M-β-x-1*) or (*M-β-x-*2), though we can eliminate the latter option due to Corollary 1.

For either of the other two rules, observe that $t = (\lambda_k x.t_1)\ t_2$. Our proof obligation then is that with

- $E \simeq \Sigma$
- $\langle e_1, \mathcal{S}\rangle \simeq \langle t_1, \eta, \mathcal{T}\rangle$
- $\langle\overline{e_2}, \mathcal{S}\rangle \simeq \langle\overline{t_2}, \eta, \mathcal{T}\rangle$
- $E\{x \mapsto e_2\} \vdash \langle e_1, \mathcal{S}\rangle \Downarrow \langle e_1', \mathcal{S}'\rangle$
- $\Sigma\{x \mapsto t_2\} \vdash \langle t_1, \mathfrak{E}(k), \mathcal{T}\rangle \downarrow \langle t_1', \eta', \mathcal{T}'\rangle$

such that

- $\langle e_1', \mathcal{S}'\rangle \simeq \langle t_1', \eta', \mathcal{T}'\rangle$, and
- η' is terminal with respect to t_1'.

This requires us to do two inductive proofs:

- We have to show that the term result preserves ($\simeq$), i.e., $e : t$ evaluates to $e' : t'$, and
- We have to show that all effects are fully consumed (except for the case of raise).

Specifically, we inductively show that for all e with $e : t$ and $e : \eta$, with

- $E \simeq \Sigma$
- $\langle e, \mathcal{S} \rangle \simeq \langle t, \eta, \mathcal{T} \rangle$
- $E \vdash \langle e, \mathcal{S} \rangle \Downarrow \langle e', \mathcal{S}' \rangle$
- $\Sigma \vdash \langle t, \mathfrak{E}(k), \mathcal{T} \rangle \downarrow \langle t', \eta', \mathcal{T}' \rangle$

we have

- $\langle e', \mathcal{S}' \rangle \simeq \langle t', \eta', \mathcal{T}' \rangle$
- η' is terminal with respect to t'.

which proves our proof obligation towards our outer induction.

We achieve both by induction over the derivation via ($\Downarrow$). First, observe that function bodies consisting of value terms — functions, constants, and particularly exceptions (cf. Figure 6.3, rule (*E-raise*)) trivially have this property. We now consider all other rules from Figure D.1 in turn.

(*var*). Our induction assumptions give us $E\{x \mapsto e_2\} \simeq \Sigma\{x \mapsto t_2\}$, so this is trivial for terms (via (*M-var*), cf. Figure D.2). Similarly, (*E-var*) (cf. Figure 6.3) states that we have no effect to evaluate in this case.

(*if-t*) and (*if-f*). Choosing the right η_h in (*M-if-t*) or (*M-if-f*), we can use our induction assumptions. We note that (*E-if*) constructs effects precisely from η_h and the `ite`. If $\eta_r = \circ$, we use (*M-empty*) to eliminate the redundant $\circ$ in the resultant trace.

(*if-x*). Follows straightforwardly from our induction assumption via (*M-if-x*). Note that we may have to use (*M-x-delete*) to eliminate effects skipped due to the exception.

(β). Beta reduction follows from our outer induction hypothesis, since we recurse precisely with the effect trace specified for the closure's lambda key (i.e., the closure can not 'accidently' consume some of our side effects). The corresponding model interpreter rules are (*M-β*) and (*M-β-x*), the former of which triggers in the case that η' (as defined in our induction hypothesis) is $\circ$, and the latter in the exceptional case.

Note that we start out with a fresh read environment ϱ for the function call and preserve our local read environment across the call.

(β-*x-1*) and (β-*x-2*). Both rules directly correspond to (*M-β-x-1*) and (*M-β-x-2*).

(*print*). In Figure 6.2, (*T-print*) interprets prints as **()**. However, (*E-print*) adds them to the effect trace. In (*M-print*), we recurse into the print's term parameter, interpreting effects η_h as prefixed by (*E-print*) — this is significant if e.g. we are printing the result of a read.

(*print-x*). Analogous to our previous exception handling but applied to the (*print*) case. Note that the exception here is 'triggered' exclusively from the effect trace.

(*handle-0*) and (*handle*). Straightforward analogues to our previous exception handling. We do need to show that the θ index of the exception matches the exception handler's index, but this is obvious via induction over the expression structure (modulo function applications, which enforce the 'surrounding' θ) by following Figure 6.2.

***(ref)*.** We assume that both (*ref*) and (*M-ref*) generate the same fresh a; otherwise this step is straightforward.

***(read)*.** Observe that we generate both an effect read, $\mathtt{r}_\rho$, and a $\textbf{read}_\rho$ term, via (*E-read*) and (*T-read*), respectively. Further observe that ρ is unassigned in $\mathcal{T}.\varrho$ until 'after' we have applied (*M-e-read*), since we enter every function body with an empty $\mathcal{T}.\varrho$ and all ρ are locally fresh. Thus, we are guaranteed that (*M-e-read*) updates the read environment before (*M-read*) can extract the value (with $\mathcal{T}.\varrho(\rho)$ undefined we cannot apply the rule).

***(a)*.** Straightforward.

***(let)*.** Straightforward.

***(ref-x)*, *(read-x)*, *(a-x-1)*, *(a-x-2)*, and *(let-x)*.** Straightforward exception propagation.

□

This result shows us that the term interpreter *may* produce the correct result. Before we can show that the interpreter is confluent, i.e., that it *must* produce the correct result, we first define an auxiliary property:

We have previously defined $\triangleright$ as being associative. However, we haven't shown that this is actually sound: it is possible that this associativity might introduce divergence. In the following, we show that that is not the case.

Lemma 6. *Let* $z \in \{\textbf{()}, \textbf{raise}_\theta\}$. *For any* $\mathcal{T}$ *and* η *such that* $\langle z, \eta, \mathcal{T}\rangle \downarrow \langle t_1, \overline{\eta_1}, \mathcal{T}_1\rangle$ *and* $\langle z, \eta, \mathcal{T}\rangle \downarrow \langle t_2, \overline{\eta_2}, \mathcal{T}_2\rangle$, *we have* $t_1 = t_2$ *and* $\mathcal{T}_1 = \mathcal{T}_2$.

Proof. We show this proof by induction over the structure of η. For all anchors, determinism is obvious. `ite` and `handle` as well as most cases of $\eta_2 = \eta \triangleright \eta'$ follow directly from the induction assumption. The only interesting cases arise when $\eta' \in \{t : \mathtt{io}, t : \mathtt{r}_\rho, t_1 \mathtt{:=} t_2\}$: In each of these cases, we have a nondeterministic choice for an η_h (cf. Figure D.2). However, all of these rules obey a small-step style semantics: it is trivial to see that, no matter how we associate ($\triangleright$), the derivation of $\langle t_i, \overline{\eta_i}, \mathcal{T}_i\rangle$ is fixed. □

Lemma 7. *Let* $t, \mathcal{T}$. *Then, for any* E, $E \vdash \langle t, \eta, \mathcal{T}\rangle \downarrow \langle \overline{t_1}, \overline{\eta_1}, \mathcal{T}_1\rangle$ *and* $E \vdash \langle t, \eta, \mathcal{T}\rangle \downarrow \langle \overline{t_2}, \overline{\eta_2}, \mathcal{T}_2\rangle$ *implies that* $t_1 = t_2$ *and* $\eta_1 = \eta_2$.

Proof. Proof by induction over term and effect trace structure. For effect traces, Lemma 6 covers all interesting cases, for terms, we note that for t, the necessary reduction rules are unambiguously prescribed either by the structure of t or by recursion.

The rules that are not solely predetermined by t fall into one of four categories:

(1) Rules that require synchrony between term and effect trace (specifically the (*M-handle*), (*M-if*), (*M-β*), and (*M-ref*) families of rules). We refer to these rules as 'synchronised rules' and note that all are unambiguous and thereby trivial.

(2) Rules that do not affect the term contents ((*M-empty*) and (*M-x-delete*)) and are otherwise known to be confluent with respect to effect traces(cf. Lemma 6).

(3) Rules that update term state ((*M-print*), (*M-e-read*), and (*M-a*)). Each such rule updates a separate component of the term state (φ, ϱ, ς, respectively), none of which are updated by any other rule. We thus only need to see that it makes no difference whether we evaluate terms or these updates first. For all such cases, first note that our (*M-β*) rules are synchronised.

For φ, note that we cannot 'read' the print output, rendering this case trivial.

For ϱ, note that each read index is locally uniquely assigned. (*M-β*) expressly preserves this property. This means that if we haven't 'executed' the proper read effect yet, our (*M-read*)) rule cannot apply, enforcing ordering.

For ς, note that each explicit read happens via (*M-e-read*) and is therefore ordered before any assignment.

(4) Rules that may 'nondeterministically' raise exceptions (specifically Rules (*M-print-x*), (*M-e-read-x*), (*M-a-x-0*), and (*M-a-x-1*)). These rules are limited in when their exceptions are applicable. Recall that each exception is tagged with the ID of the handler it is contained in, or with ♦ if not contained in an exception handler. According to (*M-handle*) we may only handle an exception with the 'correct' handler key; similarly, (*M-β-x*) and our notion of effect traces being terminal requires exceptions to be explicitly tagged with the ♦ handler key. Since exceptions are ordered among effects, we are thereby guaranteed the correct $\mathcal{T}$; the result term, in turn, must either be an exception or whatever the appropriate handler evaluates to.

□

Theorem 2. *Let* $\mathcal{T} = \langle \emptyset, [], \emptyset \rangle$ *and* $\mathcal{S} = \langle \emptyset, [] \rangle$. *Let* $E = \emptyset$ *and* $\Sigma = \emptyset$.

If $e = (\lambda x.e_1)\textbf{()}$ *is a closed expression, let* $\mathfrak{E}$ *be the effect model for* e *with respect to a term* t, *where* $e : t$. *Let* $E \vdash \langle e, \mathcal{S} \rangle \Downarrow \langle \overline{e'}, \mathcal{S}' \rangle$ *and* $\langle t, \circ, \mathcal{T} \rangle \downarrow \langle \overline{t'}, \eta', \mathcal{T}' \rangle$. *Then* $e : t$ *and* $\mathcal{S}' \simeq \mathcal{T}'$.

Proof. By Lemmata 5, 7, since $\mathcal{S} \simeq \mathcal{T}$ and $E \simeq \Sigma$. □

D.1 Program model equivalences preserve results

Next we consider all term and effect equivalences that we permitted in Figure 6.7. Examining each in turn, we show that they preserve the evaluation result.

To understand that this is correct for effects, we first need to show that our definitions in Figures 6.4 and 6.5 guarantee that the type inference-based ($:_e$) relation conservatively approximates our precise effect analysis (Figure 6.3):

Lemma 8. *If* $E \vdash e :_e \circ$ *via effect type inference, then the precise inference also types* $\theta, E \vdash e : \circ$.

We can now show our equivalence theorem:

Theorem 3. *Lemma 7 still holds under the equivalences from Figure 6.7 and Figure 6.7.*

Proof. Rule (TE-β) requires us to eliminate function application effects from the effect trace via (EE-app-○). Thus, the body of the function in question must be pure; then, the 'usual' rules about capture-free substitution hold. The capture-free substitution here is equivalent to our environment-based lookup via (*M-β*) and its exceptional variants. The only complication arises if t_2 is effectful, but in that case the effect trace must also be updated and the program will only be equivalent if our (EE-) family of equivalence rules applies.

- Analogous reasoning applies to (TE-if-t) and (TE-if-f).

- As we indicated, (EE-○) eliminates a function application without effects; we see from Lemma 8 that this is sound.
- Rule (EE-if-○) is analogous.
- Rules (EE-if-t) and (EE-if-f) are again straightforward by (*M-if-t*) and (*M-if-f*).
- Rules (EE- ○ -l) and (EE- ○ -r) are cleanup rules, corresponding to (*M-empty*).
- Rule (EE-read) allows re-ordering of read effects. Since each read effect has a (locally) unique read index, such re-ordering is safe.
- Rule (EE-ioa) allows re-ordering a print and an assignment. Both affect independent parts of the program state, making this safe.
- Rule (EE-ref) allows re-ordering fresh reference allocation with anything other than an exception. This is safe because each reference allocation creates a fresh reference and such references interfere only with reads and assignments. However, reads and assignments need the address value (a) which they can only obtain from $\mathbf{ref}_i$ which, in turn, is synchronised to a_i with a locally unique address ID i.
- Rules (EE-if-t) and (EE-if-f) are obvious from our evaluation rules (*M-if-t*) and (*M-if-f*).

□

In summary, our program model guarantees full behaviour preservation, with two exceptions:

- We do *not* guarantee that the program's termination behaviour will be preserved, and
- We do *not* validate that any user-specified axioms are correct.

The second point is an intrinsic limitation in any system that accepts user-specified axioms. Note that there are existing approaches towards validating various kinds of specifications, including automatic test generation and checking [DF94, CTCC98], concurrent program execution and specification interpretation [HRD08], and proof of correctness through a proof assistant [Pau94, Cri95]. We consider such validation to be outside the scope of this work.

Appendix E

Default congruence specifications for our SML prototype

Below is the default family of congruences that our system exposes, given in our specification language.

```
(* simple arithmetic rules *)

axiom  $A + $B == $B + $A
axiom  ($A + $B) + $C == $A + ($B + $C)
axiom  $A * $B == $B * $A
axiom  ($A * $B) * $C == $A * ($B * $C)
axiom  $A + 0 == $A
axiom  $A * 1 == $A
axiom  $A * 0 == 0
axiom  $A - 0 == $A
axiom  $A div 1 == $A

axiom  ignore ($X) == ()


congruence Seq
  domains V, L

  axiom  vector ($L) #> $L
  axiom  $V #> Vector.foldr op:: nil $V

  axiom $V #> $L ==> Vector.length ($V) == List.length ($L)
  axiom $V #> $L ==> Vector.sub ($V, $I) == List.nth ($L, $I)
endcongruence

(* vectors *)
axiom $X == Vector.concat [$X]
axiom Vector.foldl $OP $V (Vector.concat ($X :: $XS))
      == Vector.foldl $OP (Vector.foldl $OP $V $X ) (Vector.concat $XS)


(* lists *)
axiom $L1 @ $L2 == List.concat [$L1, $L2]
axiom foldl $OP $X (List.concat ($L :: $LS))
      == foldl $OP (foldl $OP $X $L) (List.concat $LS)

axiom List.nth ((map $F) $L, $I) == $F (List.nth ($L, $I))
```

Appendix F

Attribute grammar specification of the language L0

This appendix illustrates language definitions in our attribute tree grammar specification language. The specifications are similar in style to those found in other attribute tree grammar-based languages, such as Eli.

F.1 Lexical syntax definition

Language L0 has a number of terminals. Two of these are constructed from nontrivial regular expressions: *id*s and *int*s.

```
term id = [a-zA-Z][a-zA-Z'0-9]*
term int = [0-9]+
```

F.2 Abstract syntax tree definition

Below is the AST for our language.

```
Program ::= P   : (Expr)
          .

Ty ::= INT      : 'int'
     | FUN      : (Ty) '->' (Ty)
     .

Expr ::= FN     : 'fn' id (OptTy) '=>' (Expr)
       | APP    : (Expr) (Expr)
       | NAME   : id
       | INT    : int
       | TY     : (Expr) ':' (Ty)
       .

OptTy ::= N     : (* empty *)
        | TY    : ':' (Ty)
        .
```

Note that each rule for each nonterminal may have a label (such as INT and FUN for nonterminal Ty). We use these labels later as part of our specification language.

F.3 Concrete syntax definition

The concrete syntax disambiguates our abstract syntax. In the current revision of our tool, the concrete syntax can be omitted, but if it is not, the programmer has to give the full concrete syntax, even for nonambiguous rules.

Note that each concrete syntax nonterminal must specify which AST nonterminal it matches.

```
Program : Program ::= (Expr)
                    .

Ty : Ty ::= (Ty0) '->' (Ty)
          | (Ty0)
          .

Ty0 : Ty ::= 'int'
           | '(' (Ty) ')'
           .

Expr : Expr ::= 'fn' id (OptTy) '=>' (Expr)
              | (Expr0)
              .

Expr0 : Expr ::= (Expr0) (Expr1)
               | (Expr1)
               .

Expr1 : Expr ::= int
               | id
               | (Expr1) ':' (Ty)
               | '(' (Expr) ')'
               .

OptTy : OptTy ::= (* empty *)
                | ':' (Ty)
                .
```

F.4 Static semantics

The static semantics describes both name analysis and type inference, using the TypeInference plugin. Furthermore it uses the ShowFresh and Unparse plugins to visualise types with fresh variables.

```
section "Name analysis"

infer forward-chain env : id -> Expr$FN
      local Expr$NAME.origin : Expr$FN
by
    (Program$P _) =>  (env-in := |->|)
    (self as Expr$FN (id, _, _)) => (
        env-1 := update (self.env-in, id, self);
        env-out := self.env-in
    )
    (self as Expr$NAME (id)) => (origin := find (self.env-in, id,
                                  "Unbound identifier: ‘%s’" % id))
end
```

First, note the two declarations inside infer:

For name analysis, we *forward-chain* an environment env, defined as a finite map from ids to AST locations: we map each identifier to the function abstraction that defines it. Note that we specifically type the kinds of nodes we may map to, as production Expr$FN, meaning nonterminal Expr, rule FN.

We then resolve names into a local attribute origin for each production Expr$NAMEpp.

Next are three rules that define how we compute these attributes. The first rule is

```
(Program$P _) =>  (env-in := |->|)
```

Meaning ‘for any program, the starting value for the forward chain is an empty map’. Here, the construct to the left-hand side of the double arrow => represents a pattern match. On the right hand side of the arrow we express all actions that we should execute for a node that matches the rule: here, we state that any node that is of the production Program$P (i.e., a top-level program node), set that node’s attribute env-in to the empty map.

We can nest such pattern matches and directly refer to the atttributes of each node. For example, consider the next rule:

```
(self as Expr$FN (id, _, _)) => (
    env-1 := update (self.env-in, id, self);
    env-out := self.env-in
)
```

This rule matches function abstractions with an identifier of id. Note that the rule labels its local node as self in the pattern match.

Here, env-in is again the ‘input environment.’ For any chain with the name *c*, we introduce a number of attributes: *c*-in is the environment passed into the current node, *c*-out is the environment passed out of the node, and *c-i* represent local intermediate results. By default, we chain these environments forward unchanged in a depth-first traversal, though the user can override each step of the traversal, if needed. To facilitate this, each intermediate result *c-i* is the *c*-out node of the *i*th child of the current node, and simultaneously the input to the $i + 1$st child, copied to that child’s *c*-in, by default.

What our first assignment here says, then, is that the environment that we pass to the function’s body (and, incidentally, to the optional type annotation) is the same as the outer environment (env-in), except that we map the identifier id to the local node. The second assignment then ensures that this identifier mapping does not persist outside of the body of this function abstraction: the environment that we pass back to our parent node is again the same that we had passed in.

The final rule is now straightforward:

```
(self as Expr$NAME (id)) => (origin := find (self.env-in, id,
                              "Unbound identifier: ‘%s’" % id))
```

here, we compute the local attribute origin through a lookup in the environment. Function find may fail, if id is not mapped to anything; in that case, we issue a 'fatal' inconsistency (one that cannot be accepted as behavioural change).

Type inference follows the same general scheme, except that we make use of a plugin mechanism to visualise and check types. Appendix G, the documentation to our attribute tree grammar tool, details these mechanisms.

```
plugin FTYShow = ShowFresh ( ty = tyvar;
                             prefix = "'";
                             suffix = "";
                             range = "a-zA-Z"; )
freshtype tyvar

plugin SH = Unparse (ty = ty; oparen="("; cparen=")"; )
datatype ty     = INT               %SH-format="int"                        %TI-literal
                | FUN : ty * ty     %SH-infixr=1    %SH-format=" -> "       %TI-cons="-+"
                | VAR : tyvar                                               %TI-cons="v"

section "Type analysis"

infer local Ty.ty : ty
by
    (Ty$INT)            => (ty := INT)
    (Ty$FUN (l, r))     => (ty := FUN (l.ty, r.ty))
end


plugin TI = TypeInference
 (ty = ty;
                var-ty = tyvar;
                error-msg-mismatch = "Type mismatch: [%s] vs [%_%s]";
                error-msg-no-subtype = "Incompatible types: [%s] vs [%_%s]";
                error-msg-no-common-subtype = "Incompatible types: [%s] vs [%_%s]";
                error-msg-no-common-supertype = "Incompatible types: [%s] vs [%_%s]";
                error-msg-circularity = "Circular type: [%s]";
               )
section "Type inference"

infer forward-chain cset : TI-cset
      inherited varmap : TI-var-map

      local Expr.pre-ty : ty
      local Expr.ty : ty
      local Expr$FN.var-ty : ty
      local OptTy.ty : ty
by
    (self as Program$P _) =>  (cset-in := TI-cset ("function");
                               varmap := TI-solve-cset (self.cset-result))

    (OptTy$N)           => (ty := VAR (fresh (tyvar)))
    (OptTy$TY (ty))     => (ty := ty.ty)

    (self as Expr$FN (id, ty, expr)) => (
            var-ty := ty.ty;
            cset-1 := self.cset-0; (* workaround for
                                    * chains-on-rules-with-embedded-terminals bug *)
            pre-ty := FUN (self.var-ty, expr.pre-ty)
    )
    (self as Expr$APP (lhs, rhs)) => (
            pre-ty := VAR (fresh (tyvar));
            cset-out :=
                TI-constrain-eq (self.cset-result,
                                 TI-schema (self.cset-result, lhs.pre-ty), "function",
                                 TI-schema (self.cset-result, FUN
                                           (rhs.pre-ty, self.pre-ty)),
                              "expected")
    )
    (self as Expr$NAME (id)) => (
            pre-ty := self.origin.var-ty
    )
    (self as Expr$INT (_)) => (
            pre-ty := INT
    )
    (self as Expr$TY (expr, ty)) => (
            pre-ty := ty.ty;
            cset-out := TI-constrain-eq (self.cset-result,
                                         TI-schema (self.cset-result, expr.pre-ty), "actual",
                                         TI-schema (self.cset-result, self.pre-ty), "constraint")
    )

    (self as Expr _) => (ty := TI-map-ty (self.varmap,
                                          TI-type (self.cset-result, self.pre-ty)))
end
```

This language did not include a dynamic semantics. At present, the dynamic semantics only allows users to determine specific properties that they want to be part of the desired program model.

Appendix G

Attribute grammar specification language (v0.1.0)

This appendix contains a copy of the current (as of this writing) documentation for our attribute grammar tool.

```
==================================================
Overview
========

sml-atg is a programming language processing tool.  It takes a

  (1) lexical specification
  (2) syntactic specification (optional)
  (3) AST specification (doubles as syntactic specification if the
      syntactic specification is missing)
  (4) static semantics specification
  (5) dynamic semantics specification (currently very rudimentary)

and generates SML code that facilitates the following:

  (1) lexing
  (2) parsing
  (3) AST representation
  (4) static semantics computation
  (5) AST transformations, including static semantics updates
  (6) automatic validation of relevant semantic properties
  (7) unparsing

Eventually, it will also take in transformation specifications and
generate proof requirements for ensuring that the transformations are
sound.

==================================================
General
=======

Each of the inputs is presented in a separate file and follows
separate syntactic conventions.  Lexically, the following rules apply:
```

```
- Identifiers are either entirely symbolic (as per the SML standard)
  or are made up of alphanumerics, underscores, dashes, and primes,
  beginning with a character
- Comments begin with a '(*' and terminate with a '*)'.  Comments may
  be nested and placed anywhere within whitespace.

==================================================
Lexing
======

Lexemes are drawn from two sources:  from the lexer specification, and
from the concrete syntax specification (which may be automatically
derived from the AST, see below).

(1) Lexemes from the syntactic specification are extracted from
    verbatim terminals (usually used for keywords and builtin
    operators) given as part of the syntactic specification.  Such
    lexemes match fixed strings, i.e., they are insufficient for
    numeric literals or identifiers.

(2) Other lexemes are specified as part of the lexer specification.

Lexemes from the syntactic specification override lexemes from the
lexer specification.  This captures common practice outside of
FORTRAN, wherein keywords are excluded as valid identifiers.

Comments and whitespace are currently hardwired to SML conventions.
This should be fixed in a future version (shouldn't be hard, except
for making sure that we still properly track line numbers in multiline
comments.  Maybe we'll just allow ``single line comment regexp'' and
``multiline comment start/end'' regexps, with a specification
indicating whether multiline comments may be nested.).

The Lexer Specification
-----------------------

Lexer specifications are given in `.lexer' files.  Lexer
specifications are the sole exception to the comment rule:  comments
are only permitted before the beginning of the specification.  The
lexer specfcation contains a list of `terminal' definitions, as
follows:

entry ::= 'term' id '=' (regexp)
        .

where (regexp) follows the usual syntactic conventions:
```

```
regexp ::= CHARACTER : char
         | WILDCARD : '.'
  | CHOICE : (regexp) '|' (regexp)
  | PAREN : '(' (regexp) ')'
  | ZERO-OR-ONE : (regexp) '?'
  | ZERO-OR-MORE : (regexp) '*'
  | ONE-OR-MORE : (regexp) '+'
  | CHARSET : '[' char ... char ']'
  | NEG-CHARSET : '[^' char ... char ']'
  | REF : '{' id '}'
 .
```

Here, CHOICE has the least precedence, while the quantifiers bind most tightly.

The CHARSET and NEG-CHARSET constructions allow the definitions of character sets and negated character sets, respectively. A CHARSET matches precisely one character, if that character is one of the chars listed within. As a special exception, the character '-' may be used to denote ranges, as in ‘[a-z]’ which will match all lower-case ASCII characters. To denote a literal '-', escape it or list it as the first or last character of the set. A NEG-CHARSET matches precisely the characters the corresponding CHARSET would not match, e.g., ‘[^a-z]’ matches all characters EXCEPT for lower-case ASCII characters.

The REF construct refers to (syntactically) previously defined terminals, such as

```
term digit = [0-9]
term int = {digit}+
term real = {int}\.{int}
```

Characters (char) are single alphanumeric characters or any other characters preceeded by a backslash, with the following exceptions:

- ‘\t’ is the tabulator character (ASCII code 9)
- ‘\n’ is the newline character (ASCII code 10)
- ‘\r’ is the carriage return character (ASCII code 13)
- ‘\f’ is the form feed character (ASCII code 12)

The form ‘\[0-9][0-9][0-9]’ is also permitted for specifying characters by their ASCII ordinal (e.g., ‘\032’ is a blank).

Note that the lexer generator will only generate code to recognise

```
lexemes that are used in the concrete syntax.  This avoids superfluous
tokens for terminals that are only meant as auxiliaries.


Examples
--------
3.lexer:
--------
term alpha = [a-zA-Z]
term alnum = ({alpha}|[_'0-9])
term name = {alpha}{alnum}*
term intlit = -?[0-9]+

sml.lexer:
----------
term string = \"([^\\\"]|\\[\ \t\n\r\012]+\\|\\[^\ \t\n\r\012])*\"
term digit = [0-9]
term decint = [0-9]+

term hexint = [0-9a-fA-F]+
term alnum = [0-9a-zA-Z'_]
term sym = [-!%&$#\+/:<=>\?@\\~`\^\|\*]

term int-lit = ~?({decint}|0x{hexint})
term word-lit = 0w({decint}|x{hexint})
term string-lit = {string}
term char-lit = #{string}
term id-nonprime = ([a-zA-Z]{alnum}*)|{sym}+
term id-prime = '({alnum}+|{sym}+)
term real-lit = ~?({decint}\.{decint}|{decint}[Ee]
                ~?{decint}|{decint}\.{decint}[Ee]~?{decint})


==================================================
AST
===

The AST specification is mandatory and placed in a `.ast' file.  The
AST specification describes the AST that both the static analysis and
the first pass of the semantic analysis operate on.  The AST need not
equal the concrete program syntax and will typically be simpler.

The AST specification consists of sets of nonterminal rulesets of
the form

nonterm-ruleset ::= id '::=' rules '.'
  .
```

```
rules ::= rule
        | rules '|' rule
.

rule ::= ANONYMOUS : rhs
       | NAMED : id ':' rhs
       .

rhs ::= NONTERM : '(' id ')'
      | TERM : id
      | VTERM : vid
      .
```

where a ‘vid’ is a string enclosed in single or double quotes.
Each nonterminal ruleset thus consists of a list of rules. Each rule in turn may be anonymous or named, and has a right-hand side.
Consider the nonterminal ruleset

```
N ::= (* empty *)
    | id
    .
```

This ruleset has two rules, both anonymous: the first rule matches the empty production, the second rule matches the production of a single terminal ‘id’ (which must be part of the lexer specification).

Here is another nonterminal:

```
R ::= id
    | (R) ',' id
    .
```

This nonterminal ruleset describes a nonempty comma-separeted list of ‘id’s. Note that references to nonterminals are enclosed in parentheses. The ',' here is a verbatim terminal: such terminals are implicitly added to the lexical structure of the specified language.

Finally, named rules look as follows:

```
N' ::= EMPTY : (* empty *)
     | ID : id
     .
```

N' matches the same grammar as N and are interchangeable. However, N' provides a name for each rule; these names are used within the static and dynamic semantics to distinguish the different cases for each nonterminal. For anonymous rules, the system automatically generates names according to the following scheme: for a nonterminal N, the syntactically first rule has the name N0 (unless it is named), the second rule has the name N1 (unless it is named) and so on. Since modifications to the AST may shuffle these names around, anonymous rules should be used sparingly.

The first nonterminal specified in the AST is considered to be the start terminal.

Since the AST must match the concrete syntax and the concrete syntax is translated into an LALR(0) parser specification, the AST should strive to be left-recursive whenever practical.

```
==================================================
Syntax
======
```

The concrete syntax may be specified in a '.syntax' file. In the absence of such a file, the system uses the AST specification directly as concrete syntax. However, for most languages, having a separate syntax is useful for keeping the AST simple:

- Some constructions, such as ''possibly empty list'' require more productions in the concrete syntax (four in the example) than we need for the AST (two in the example).
- The LALR(0) restriction of the input grammar may force some changes to the syntax that would unneccessarily complicate the AST further.

We refer to the concrete syntax tree as CST for short.

The CST, if present, must provide a full syntactic specification of the language, even in cases where AST and CST coincide. The CST specification consists of a sequence of nonterminal rulesets similar to the AST rulesets, but following the syntax

```
cnonterm-ruleset ::= id ':' id '::=' crules '.'
   | id '::=' crules '.'
   .

crules ::= crule
        | crules '|' crule
 .

crule ::= ANONYMOUS : rhs
       | NAMED : id ':' rhs
       .
```

with rhs defined as for the AST.

To see this in practice, consider the follwing specification of a comma separated list:

```
AST:

  ASTList ::= NIL : (* empty *)
            | CONS : (ASTList) id
    .

CST:

  CSTList : ASTList ::= (* empty *)
                      | (CSTList1)
      .

  CSTList1 : ASTList ::= id
                       | (CSTList1) ',' id
       .
```

Here, the AST uses two named rules for the definition of ‘ASTList’, while the CST provides an analogous definition ‘CSTList’. Note that both ‘CSTList’ and ‘CSTList1’ are annotated with ‘: ASTList’, which indicates that they should be interpreted as producing an ‘ASTList’ in the AST. When parsing, the system will automatically translate the CST matches into an appropriate AST structure and annotate the AST with sufficient information to recover the CST.

Due to the matching algorithm used, the AST must be a homomorphic image of the CST with element ordering preserved. This restricts the ways in which the two may diverge; in particular, a left-recursive CST will always produce a left-recursive AST.

Sometimes the system may be unable to distinguish different cases automatically. For example,

```
AST:

  ASTX ::= REF : id
         | VAL : id
 .

CST:
  CSTX : ASTX ::= '&' id
             | id
.

Here, the two AST rules look identical, so the system cannot
distinguish which case to use and will produce an error message.
The CST allows explicit rule annotation to distinguish in such cases:

CST:

  CSTX : ASTX ::= REF : '&' id
             | VAL : id
.

(where 'REF' and 'VAL' directly refer to the rules from the AST).

Similarly, it is possible to specify that a rule should be a
'decoration' rule only, as often used for parentheses:

  CSTX : ASTX ::= REF : '&' id
             | VAL : id
| _ : '(' (ASTX) ')'
.



==================================================
Static Semantics
================

Properties
----------

* local
* inherited
* synthesised
* forward-chain
* backward-chain

forward-chain and backward-chain properties describe depth-first tree
traversals, with forward-chain properties going left-to-right and
backward-chain properties going right-to-left.  Each chain declaration
has the form

  forward-chain <name> : <ty>
or
```

```
  backward-chain <name> : <ty>

Let <name> be C.  Then the following properties will be available on
ALL AST nodes:
* C-in
* C-out
* C-result
and for an AST node with n+1 AST children,
* C-0
  ...
* C-n

  The meanings of each of those properties is as follows:
C-in is the incoming value (flowing in from above).  C-out is the
outgoing value (flowing back to the parent node).  C-result is the
result computed for this AST node; by default, it is copied directly
to C-out.

  The various C-k (with k a natural number) are precisely the values
that flow INTO the child at index k.  Thus, the values that flow OUT
OF the child at index k are C-{k+1} (for forward chains) and C-{k-1}
(for backward chains)-- unless we are already at the last node.  In
that case, the outflowing value is again C-result.


Builtin operations
------------------
Most builtin operations are used like regular functions.  The
exceptions to this rule are infix operators, which (as their name
implies) are placed in between their two parameters.  All infix
operators are left-associative and belong to one of two levels of
precedence, where ‘andalso’ and ‘orelse’ alone form the
lower-precedence level.

Logical operations:
-------------------
Value constructors:
- true : bool
- false : bool

(0)  (infix) = : ’a * ’a => bool

  Compares two values for equality.  The values must not be or contain
functions, though finite maps are permitted.

(1)  (infix) <> : ’a * ’a => bool
```

```
  Compares two values for inequality.  This function is the inverse of
(=) above.

(2)  not : bool => bool

  Negates a boolean value.

(3)  (infix) andalso : bool * bool => bool

  Builds the conjunction of two values, using short-circuit
evaluation (this is relevant for failure).

(4)  (infix) orelse : bool * bool => bool

  Builds the disunction of two values, using short-circuit
evaluation (this is relevant for failure).


List operations:
----------------
Value constructors:
- :: : 'a * 'a list => 'a list
- [] : 'a list

(0)  first : 'a list => 'a

  Extract the first element of a list, or abort with an inconsistency
if there is no first element.

(1)  rev : 'a list => 'a list

  Reverses a list

(2)  length : 'a list => nat

  Determines the length of a list


Natural numbers operations:
---------------------------

(0)  parse-nat : [0-9]+ -> nat

  Represents a digit or sequence of digits as a natural number.


String operations:
```

```
------------------
String literals can be specified in between double quotes, with the
usual rules for escaping.

(0)  show : 'a => string

  Maps an arbitrary object to a string.  Not permissible for function
types.

(1)  (infix) % : string * 'a => string

  Formats a value into a string.

  s % msg

  will stringify ‘msg’ and substitute it for the first occurrence of
a formatting operator in ‘s’.  The format for such formatting
operators is as follows:

  %% Literal percent sign
  %s Default substitution (using whichever ‘show’ function
   is in use for the value in question)
  %_ Ignore
  %Sseq; Substitution with special formatting for sequences
   (sets, arrays):  The ‘seq’ part is used as separator
between entries.  Use ‘%;’ to denote a literal
semicolon.
  %Mseq;arrow; Special formatting for maps.  The ‘seq’ part is again
   the separator, the arrow part is the mapping arrow
(left-to-right) for map elements.
  %:pfx;sfx;C specifies that ‘pfx’ and ‘sfx’ should be printed as
   prefixes/suffixes for the formatting specification in
C (which should be given without the initial %) IF AND
   ONLY IF the formatting of C yields a nonempty string


Set operations:
---------------
(0)  {} : 'a set

  Constructs an empty set.

(1)  insert : 'a set * 'a => 'a set

  Inserts a value into a set.

(2)  contains : 'a set * 'a => bool
```

```
  Tests whether a value is contained in the specified set.

(3)  union : 'a set * 'a set => 'a set

  Constructs the union of two sets.

(4)  intersection : 'a set * 'a set => 'a set

  Computes the intersection of to sets.

(5)  set-difference : 'a set * 'a set => 'a set

  Computes the difference between two sets.  Specifically,
'set-difference (a, b)' computes the set of all elements in 'a' but
not in 'b'.


Finite Map operations:
----------------------
(0)  |->| : 'a -> 'b

  Constructs an emty finite map.

(1)  find : ('a -> 'b) * 'a * string => 'b

  Looks up a value in a finite map.  If the value is not present, it
fails with an error message indicated by the string.

(2)  update : ('a -> 'b) * 'a * 'b => 'a -> 'b

  Updates the map.

(3)  prepend-map : ('a -> 'b) * ('a -> 'b) => 'a -> 'b

  Merges two maps into one.  'prepend-map (l, r)' will make 'r' a
'backup' map in case lookups fail for 'l'.  For example,

  'prepend-map (insert (|->|, "x", "new"),
                insert (|->|, "x", "old"))'

is identical to

  'insert (|->|, "x", "new")'

(4)  dom : ('a -> 'b) -> 'a set
```

```
  Constructs a set of the map’s domain.


Plugs:
------
(0)  plug : string * ’a -> ’a

  This is a projection of the second value.  However, this operation
will make a ‘plug’ (a ref variable) of the specified name available to
the calling infrastructure, thereby allowing the value to be altered.
This is mostly useful for setting initial environments.  The name
must be a literal string.

Failure:
--------
(0)  fail : string => ’a

  Explicitly aborts computation for the specified reason.


==================================================
Plugin mechanism
================

  The set of attribute grammar functions can be extended through
plugins, some of which are provided by default.  The possible effect
of plugins is highly restricted: if we allow plugins to change the
AST in arbitrary ways, it becomes impossible to guarantee proper
evaluation, restructuring, or reasoning.

  We distinguish between two classes of plugins:

(1) Structural plugins
(2) Evaluation plugins
(3) Auxiliary plugins

  Structural plugins have the capacity to modify the AST.  Their
principal purpose is to perform parse refinement:  by using additional
semantic information, they can adjust the AST (without changing the
number or order of nodes) to refine the type of a node or to introduce
additional structure.  For example, some languages permit
user-specified fixity annotations:  the ‘FixityParse’ plugin module
then allows replacing one nonterminal that provides a list of items
with one that provides an appropriate fixity parse.

  Since such transformations change the nature of individual nodes,
they must be performed as early as possible.  Thus, all structural
```

plugins are executed BEFORE any other evaluation, though they may rely on arbitrary AST computations before that.

Evaluation plugins, on the other hand, merely provide additional evaluation operations. However, their capacity to use metaprogramming implies that they can provide nontrivial operations (such as type inference) that might not be feasible otherwise.

Auxiliary plugins override 'default' behaviour, specifically pretty-printing and equality.

Naming conventions

Each plugin module must be instantiated to a specific module name. By convention, such instances (like the modules themselves) use camel notation. All functions, attributes and tags introduced by a module begin with this module name. For example, if we instantiate TypeInference to TI, then its 'fresh-env' function assumes the name 'TI-fresh-env'. This allows modules to be used in multiple different configurations.

Parameter passing

Plugins can take the following kinds of parameters during instantiation:

- string: character strings
- nat: natural numbers (also used to encode flags)
- attribute: attributes, local or not
- nonterm: nonterminal names
- datatype: datatypes
- freshtype: freshtypes

The corresponding syntax is illustrated in the following:

```
plugin <id> = <plugin-name> ( <name> = <value>;
                     ...
                     <name> = <value> ; )
```

More precisely:

```
Plugin ::= 'plugin' id '=' id '(' (PluginArglist) ')'
         .
```

```
PluginArgList ::= /* empty */
        | (PluginArg) (PluginArgList)
.

PluginArg ::= id '=' (PluginValue) ';'
    .

PluginValue ::= id
      | nat
      | string
      .
```

  Furthermore, datatype constructors can be annotated, with each
annotation taking a string, a number, or no parameter at all (acting
as a flag).
  We omit the corresponding syntax and instead give only an example:

```
datatype t =
     STRING           %TI-literal   %PP-format="string"
   | PRODUCT : t * t  %TI-cons="++" %PP-format="*"     %PP-infixl=4
   | VAR : id   %TI-cons="v"  %PP-format="%s"
```

  Note that the possible number of annotations on a constructor is
unbounded.

```
Built-in plugins
================

FixityParse (structural plugin)
-------------------------------
```

  Fixity parser.

  Attributes:

    - input-nt : nonterm
      The nonterminal we use as input. After fixity parsing, no node
      will be an instance of this nonterminal anymore.

    - output-nt : nonterm
      The nonterminal we use as output. Errors about this node not
      being produced by the parser are suppressed.

    - precedence-attr : attribute
      A ‘nat’ attribute that gives precedence. It’s assumed that

```
    ‘greater’ means ‘binds more tightly’.

  - associativity-ty : datatype
    A datatype that describes fixity behaviour (see annotations).

  - associativity-attr : attribute
    A ‘fixity’ attribute that determines whether a given node should
    be treated as nonassociative, left-associative, or
    right-associative.

  - invert-precedence : nat
    If nonzero, this changes precedence behaviour such that
    ‘greater’ in ‘precedence-attr’ means ‘binds less tightly’.

  - join-precedence : nat
    The ‘application’/‘sequencing’ rule has its own precedence,
    specified here.  The value specified here must be one more
    than the actual precedence (to permit use of precedence -1).

  - error-msg-missing-lhs : string
    Error message to indicate that an infix operator is missing its
    left-hand side argument

  - error-msg-missing-rhs : string
    Error message to indicate that an infix operator is missing its
    right-hand side argument

  - error-msg-precedence-conflict : string
    Error message to indicate that two operators with identical
    associativity but inverse associativity are both trying to
    ‘claim’ a value

  - error-msg-infix-pair : string
    Error message to indicate that two infix operators are next to
    each other

Annotations (on ‘fixity’):

  - %left
    If ‘fixity’ assumes a value tagged with %left, the operator is
    assumed to be left-associative.

  - %right
    If ‘fixity’ assumes a value tagged with %left, the operator is
    assumed to be right-associative.

Unannotated ‘fixity’ states are considered to be non-associative.
```

```
ShowFresh (auxiliary plugin, overrides ‘show’)
----------------------------------------------

  Visualisation override for ‘show’, for freshtypes.

  Attributes:

    - ty : freshtype
      The particular freshtype that should be visualised

    - prefix : string
      A prefix for every generated name

    - suffix : string
      A suffix for every generated name

    - range : string
      A range expression, denoting the characters that should be
      considered.  After running out of characters  of this range, the
      system will append characters from the range.  Ranges consist
      of sequences of characters that should be used, but may be
      abbreviated using a ‘dash’ sign.

      Example: range="a-zA-Z"

  If used on an individual freshtype instance, the result will always
be the same.  However, when distinct freshtype instances are used in
the same expression, the system ensures that all generated names are
distinct (even between disjoint freshtypes).

Unparse (auxiliary plugin, overrides ‘show’)
--------------------------------------------

  Unparsing-based ‘show’ implementation for datatypes

  Attributes:

    - ty : datatype

    - oparen : string
      Character or string to use to represent the notion of an opening
      parenthesis.  default: "("

    - cparen : string
      Character or string to use to represent the notion of a closing
      parenthesis.  default: ")"

  Annotations (on ‘datatype’):

    - %format
      The string token to represent this datatype constructor,
      formatted as needed.  The default is "%s", repeated for each
      argument.  This is refined by the following options:

    - %prefix
      Specifies that the token should act as a prefix on its first
      parameter (later parameters are ignored).  An optional parameter
      encodes precedence (higher -> binds more tightly).

    - %suffix
      Specifies that the token should act as a suffix on its first
      parameter (later parameters are ignored).  Analogous to %prefix.

    - %infixl
      Specifies that the token should act as infix operator on its
      first two parameters (later parameters are ignored) and is
      otherwise left-associative.  Behaviour is undefined with less
      than two parameters.

    - %infixr
      Right-associative analogue to %infixl.

    - %inline
      Indicates that the parameter is a list of entries that should be
      treated as individual children for recursive parsing.

    - %sequence
      A sequence of items separated by the format string.  Precedence
      of the separating operator can be given optionally.

    - %surround
      Requires a format string of the form ‘prefix%Sseq;suffix’.
      Formats all parameters as the specified sequence; this overrides
      the %format directive.

    - %terminal
      Use the parameter %format string stand-alone and ignore any
      children (except for formatting).

  If not specified, the default is that of ‘%suffix’.
```

```
TypeInference (evaluation plugin)
---------------------------------

  Constraint-based type inference system.

  Attributes:

    - ty : datatype

    - var-ty : freshtype
      Type of type variables

    - error-msg-mismatch : string
      Type mismatch.  Four parameters are passed into this format
      string:  type1, descr1, type2, descr2.

    - error-msg-no-common-supertype : string
      Failed to indentify common supertype.  Four parameters are
      passed in:  type1, descr1, type2, descr2.

    - error-msg-no-common-subtype : string
      Failed to indentify common subtype.  Four parameters are
      passed in:  type1, descr1, type2, descr2.

    - error-msg-no-subtype : string
      type1 is not a subtype of type2.  Four parameters are passed in:
      type1, descr1, type2, descr2.

    - error-msg-circularity : string
      A type variable circularly depends on itself.  Only one
      parameter is passed in (type1).

  Annotations (on ‘type’):

    - %literal
      Literal type.  All parameters (if any) must match exactly.

    - %cons
      Type constructor.  The parameter string’s length must have a
      length identical to the number of parameters to the constructor;
      each specified character indicates how the value should be
      treated:

      = Invariant equality:  The type checker enforces
       equality.  If the parameter recursively references
       another ‘datatype’ value, equality is interpreted as
       ‘modulo substitutability’.
```

```
Two sets, s1, s2

      + Covariant equality.  The converse to contravariant
       equality (cf. above).
Two sets, s1, s2, are covariantly s1 <: s2 iff s1 is a
       subset of s2.

      -Contravariant equality:  Same as equality, but
                whenever a subtype constraint A <: B is created, the
                value in A may have a supertype in this location
                instead.
Two sets, s1, s2, are covariantly s1 <: s2 iff s1 is a
       SUPERSET of s2.

      ? Ignore this value, but try to preserve it.


      The characters below have additional restrictions on when and
      how they can be used:

      T Inlined tuple.  This must be a ty list and the rule
       must NOT contain any additional 'ty' entries
       (including recursive maps, ty lists etc.)

      v Type variable.  This must be a var-ty, and the rule
       must consist ONLY of this declaration.

      _ Ignore this value completely.  Only valid with the
       projection/selection operator(s) (see below).

      @ Projection operator.  Only valid if all other
       parameters are marked as '_'.  This then selects the
       one parameter that should be used in place of the
       entire cons expression for purposes of type inference.

    - %domain
      This operation applies to all sets and maps in the current
      constructor and determines whether their domains may be
      extended.  Possible values are

      -Never

      + Always

      i (where 'i' is a number) iff the bool at 0-based
       index 'i' is 'true'
      ~i (where 'i' is a number) iff the bool at 0-based index
       'i' is 'false'

  Types introduced:

      cset
Constraint set

      var-map
Variable map

      ty
Encoded type (referred to as "ty'" below)

  Operations introduced:

      cset : string => cset
the string parameter is currently unused (may later be used to
indicate the cset origin)

      clone-cset : cset => cset

      solve-cset : cset => var-map

      constrain-eq : cset * ty' * string * ty' * string => cset

        constrain-eq (cset, ty1, descr1, ty2, descr2) constrains ty1
        and ty2 to be equal.  descr1 and descr2 are used in error
        messages (see above).

      constrain-sub : cset * ty' * string * ty' * string => cset

        constrain-sub (cset, ty1, descr1, ty2, descr2) constrains ty1
        to be a subtype of ty2.  descr1 and descr2 are used in error
        messages (see above).

      type : cset * ty => ty'
Pack type into an encoded type

      schema : cset * ty => ty'
Type encoding with schema abstraction of all type variables

      map-ty : var-map * ty => ty


Map (evaluation plugin)
-----------------------
```

```
  (Currently not operational)

  Recursive mapping function generator.

  Attributes:

    - ty : datatype
      The datatype we wish to map over.

  Operations introduced:

    - map : (ty => ty) => ty => ty

MapVar (evaluation plugin)
--------------------------

  Recursive variable substitution function generator.

  Attributes:

    - ty : datatype
      The datatype we wish to map over.

    - var : freshtype

  Annotations (on ‘type’):

    - %var
      Tags a constructor that must contain PRECISELY ty-var as
      parameter as a subst-able constructor

  Operations introduced:

    - map : ty-var -> ty * ty => ty

TagVar (evaluation plugin)
--------------------------

  Recursive variable tagging function generator.

  Attributes:

    - ty : datatype
      The datatype we wish to map over.

    - ty-var : freshtype
      The variable we wish to tag.
    - invert : int
      Whether to invert the ty-var set when determining tagging (see
      below).

  Annotations (on ‘type’):

    - %source
      Tags a constructor as source for the tagging.  The type of this
      constructor must either match exactly the type of the unique
      %dest.
      Note that the source will only be mapped if ALL ty-vars from
      within are in the ty-var key set (see below).

    - %dest
      Tags a constructor as destination for the tagging.  Precisely
      one dest must be specified.

  Operations introduced:

    - tag : ty-var set * ty => ty

      Tags the type.  The first parameter is the ty-var key set:
      source constructors are mapped iff all of their ty-vars are
      in this set (or not in this set, if invert <> 0).
```