Program Metamorphosis

University of Colorado at Boulder
Technical Report CU-CS 1036-07

Christoph Reichenbach*
reichenb@colorado.edu

Amer Diwan
diwan@colorado.edu

December 4, 2007

Abstract

Modern agile software engineering practices encourage programmers to refactor their code frequently. Consequently, modern integrated development environments incorporate machine support for refactoring; such machine support takes the form of automatic program transformations that atomically preserve program behaviour.

This approach to refactoring is useful, but limits the approach in two ways: first, existing refactoring tools enforce that each individual step preserves behaviour– which can be too restrictive. Secondly, such tools cannot permit behaviour-enhancing changes, even though such changes are both desirable and automable.

We present an extension to refactoring that addresses both of the above limitations, by (1) relaxing behaviour-preservation checks to occur only after sequences of transformations rather than before each individual transformation, and by (2) permitting users to accept rather than to correct some of the behavioural changes that accumulate during such transformation sequences.

We evaluate our approach, which we call "program metamorphosis", by applying it to the task of structural and nominal transformations of Standard ML programs. We find that our approach (1) can effectively assist programmers in the refactoring process whilst eliminating the aforementioned shortcomings of traditional refactoring systems, (2) can be implemented efficiently, and (3) fully subsumes traditional refactoring.

*Supported by NSF Career Grant CCR-0133457

1 Motivation

Modern programming methodologies, such as Extreme Programming [3], use *refactorings* to prepare software for impending changes or to eliminate "bad smells" in source code. Fowler *et al.* [6] defines refactorings as:

> *A change made to the internal structure of software ... without changing its observable behaviour*

To automate this process, modern IDEs such as Eclipse [15] and refactoring engines such as HaRe [9] provide machine support for refactoring. These systems implement refactorings as a pair $\langle P, t \rangle$: if the precondition P holds for a given program, the IDE can apply the transformation t to that program [8, 10]. If P does not hold, the IDE disallows the transformation. The underlying assumption is that whenever P holds, t will preserve behaviour.

This approach allows refactoring engineers to build refactorings that are sufficiently conservative to be safe. Unfortunately, it sometimes also forces them to be too conservative:

1. Many practical refactorings require multiple steps to complete. Consider moving a field and related methods via the "Move Method" and "Move Field" refactorings [6]: if the field is invisible outside of the current class, we cannot move it separately from the methods, unless we introduce auxiliary "getter" and "setter" operations that the user will later have to remember to remove again. With many fields or mutually recursive methods, the problem is exacerbated; users may find it easier to abandon the safety of refactorings in favour of faster manual editing.

   To fit refactorings such as the above into the "precondition and transformation" framework, refactoring developers must construct large and complex refactorings. However, such refactorings are harder for users to comprehend, predict, and apply.

2. By definition, refactoring must not permit observable behavioural change. This means that even program modifications that fit within the scope of what refactoring transformations can accomplish– such as renaming a method in a public library API, or eliminating or introducing calls to a logging mechanism– are not permissible. This creates a dilemma for designers of refactoring engines: should they obey the definition of refactorings, or should they maximise the utility of their systems?

We propose an alternative to refactoring that addresses both of the above limitations, without losing the advantages of refactoring.

Our approach is based on the idea of eliminating a need for preconditions, thereby permitting free-form program manipulation. To ensure behaviour preservation, we compare the *current program behaviour* against the initial program behaviour. This gives us the flexibility to perform refactorings in multiple steps.

In this process, the initial program behaviour has the effect of a desired "goal behaviour" for the transformation. We permit users to expressly update this *desired program behaviour*, thereby altering the behaviour that our system enforces.

Since our approach is strictly more powerful than refactoring, we give it a different name, *program metamorphosis*.

Our contributions are the following:

- We present program metamorphosis, a novel approach to refactoring and user-controlled behaviour evolution.
- We illustrate how our approach arises naturally out of traditional machine-supported refactoring, and how program metamorphosis is superior:
  - Program metamorphosis subsumes traditional refactoring and can act as a toolkit for implementing traditional refactoring from scratch.
  - Program metamorphosis is more flexible in that it (a) does not need to enforce behaviour-preservation after each individual transformation and (b) permits (user-mandated) behavioural evolution.
  - Program metamorphosis allows transformation developers to ignore "fixup heuristics" (Section 2.5): existing refactoring tools must often heuristically apply implicit refactorings to enable the preconditions of user-selected refactorings; this need disappears in program metamorphosis.
- We derive an abstract implementation strategy that makes our approach practical and show that it subsumes traditional refactoring.
- We describe two concrete implementation strategies and discuss our experiences with them.
- We report execution times for program metamorphosis to illustrate that the approach is practical.

The rest of this paper is organised as follows. Section 2 introduces the concept of program metamorphosis on a high level. Section 3 illustrates our approach with an example, based on a subset of SML. Section 4 describes how we can efficiently implement refactoring. Section 5 gives an extended example of the implementation mechanism, describing our prototype. Section 6 evaluates our prototype implementation. Section 7 discusses practical differences between our approach and refactoring. Section 8 reviews related work, and Section 9 concludes.

2 From Refactorings to Program Metamorphosis

Refactorings have proven to be invaluable tools for software engineers. However, refactorings are limited in a number of ways. Our approach, program metamorphosis, eliminates two of these limitations while retaining the inherent advantages that refactorings offer over manual program manipulation. First, program metamorphosis allows *behaviour evolution*. Secondly, program metamorphosis permits behaviour-preserving program manipulation with fewer restrictions than refactoring.

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.

2.1 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 properties computes all properties that are relevant to the refactoring and some check c_P determines whether the properties will guarantee behaviour preservation.

Since 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 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 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[9]) are not conservative, conservativeness (and therefore guaranteed behaviour preservation) remains the underlying motivation behind refactoring, and therefore an ideal for all refactoring tools.

2.2 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: inlining may require variable renaming to avoid name capture. If we rename first, the refactoring is behaviour-preserving, but if we inline first, it is not. 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 mutually recursive procedures from one module to another. With a "Move Procedure" refactoring, we expect this transformation to require two steps, one for each individual move. But if we move only one procedure, the procedures will not be able to call each other, and thus we will not preserve behaviour. Consequently, we must move both simultaneously to preserve behaviour.

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 [8] 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.

2.3 Towards Program Metamorphosis

Our approach, *program metamorphosis*, 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.

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,

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

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

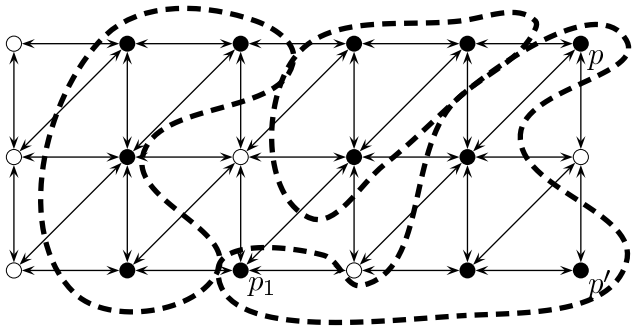


Figure 1: Program metamorphosis. Vertices represent programs; programs are well-formed (V) precisely if they are black vertices. Solid edges represent possible program transformations. Dashed lines represent the equivalence classes of ($\equiv$).

Preconditions are not normally implemented as instances of the above formula, though we can re-write them appropriately (as Section **??** points out, this does not limit our approach.)

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 (this is not completely straightforward, cf. Theorem 1). 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 1: 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 $\overline{t} = t_1 \circ \cdots \circ t_n$ such that

$$Q(\overline{t}(p)) \iff V(p) \wedge (p \equiv \overline{t}(p)) \wedge V(\overline{t}(p))$$

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 choose $\overline{t}$ to be a single transformation. If we want to refactor p to p' in Figure 1, we thus need one extra

step to reach p', and we can never reach p_1, even though it is semantically equivalent to p.

2.4 Soundness and Derivation

We can summarise our above observations as follows:

Definition 1. *A* (program) metamorphosis system *for a language L is a tuple $\langle \equiv, V \rangle$ where $(\equiv) : L \times L$ is an equivalence relation, $V \subseteq L$, and $V(p)$ whenever the program p is* well-formed.

Equivalently, we can combine the notion of a program metamorphosis system with the $\mathsf{properties}$ function that refactorings use. This function maps a program to a "program model", a set of $M \in \mathcal{M}$ of relevant program properties. For a given metamorphosis system, a metamorphosis system with $\mathsf{properties}$ is then simply a tuple $\langle \mathcal{M}, \mathsf{properties}, \equiv_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ with $\mathsf{properties} : L \to \mathcal{M}$ where

$$\begin{aligned} V(p) &\iff V_{\mathcal{M}}(\mathsf{properties}(p)) \\ p_1 \equiv p_2 &\iff \mathsf{properties}(p_1) \equiv_{\mathcal{M}} \mathsf{properties}(p_2) \end{aligned}$$

For speculative metamorphosis systems, the above is sufficient. In practice, we usually want our systems to preserve behaviour:

Definition 2. *A program metamorphosis system is* sound *wrt a language semantics $[\![-]\!]$ iff, for all programs $p, p' \in L$,*

$$V(p) \wedge p \equiv p' \wedge V(p') \implies [\![p]\!] = [\![p']\!]$$

Conveniently, we can construct metamorphosis systems from refactoring preconditions such that the metamorphosis systems are sound whenever the refactoring preconditions are conservative. Recall our earlier decomposition of preconditions:

$$P(p) \iff V(p) \wedge (p \equiv t(p)) \wedge V(t(p))$$

If we set $(\equiv) = (\equiv_{[\![-]\!]})$, where $a \equiv_{[\![-]\!]} b \iff [\![a]\!] = [\![b]\!]$, we have the "perfect" predicate for *any* refactoring. This relation is not computable, so we must choose another. If we choose not to be conservative (i.e., if we do not guarantee behaviour preservation), we may pick any relation. If we are conservative, we must pick a $(\equiv) \subset (\equiv_{[\![-]\!]})$, i.e., a conservative approximation that distinguishes some programs that would be semantically equivalent. If we are conservative, we can now show the following:

Theorem 1. *Given the decomposition of refactoring preconditions $P_1, \ldots, P_n$, we can construct a metamorphosis system that is sound if $P_1, \ldots, P_n$ are conservative, and allows at least as many transformations as $P_1, \ldots, P_n$ allow.*

Proof. Let $(\equiv_1), \cdots, (\equiv_n)$ be the equivalence relations used in $P_1, \ldots, P_n$. Then we set

$$(\equiv) = (\equiv_1) \cup \cdots \cup (\equiv_n)$$

All $(\equiv_i)$ are conservative approximations of $(\equiv_{[\![-]\!]})$, so $(\equiv)$ inherits this property. Furthermore, for any programs p_1, p_2 we have that $p_1 \equiv_i p_2$ $(1 \leq i \leq n)$ implies $p_1 \equiv p_2$. □

2.5 Multi-step Transformations in Practice

What happens if the postcondition Q fails after a sequence of transformations? Apart from requiring users to manually satisfy Q, we have several choices, listed below:

(i) *Disallowing/retracting transformations* that fail to preserve behaviour is a straightforward strategy.

(ii) *Heuristically applying supporting transformations* is another strategy that existing refactoring tools, such as Eclipse, employ, albeit for traditional refactorings. These tools predict conflicts that will happen and heuristically pre-apply other refactorings that will avoid such conflicts. The resulting heuristics may have undesired side-effects; if so, the user must undo the heuristic changes later.

While program metamorphosis is also amenable to heuristics, they are not necessary, so we do not explore this concept here.

(iii) *Searching for recovery plans* is another possible strategy: 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. In this paper, we do not investigate such planning heuristics in detail.

(iv) *Accepting behavioural change* is another strategy, unique to program metamorphosis. This option is useful in many practical scenarios, such as intentionally changing a public library API, re-ordering side effects that the user deems independent (such as log output), or even just as

an override mechanism in situations where the user wants to manually intervene because the system is lacking the transformations she needs. Whatever the reasons for accepting change, the user must explicitly permit such changes.

The most straightforward way to implement behavioural change is to override the final equivalence check. This implementation is simple but dangerous, as we override all behavioural changes at once. We propose an alternative approach, wherein users acknowledge individual behavioural changes. We describe this approach in Section 4.2.1.

2.6 Summary

In summary, we can construct a program metamorphosis system from a set of refactorings by extracting the inherent equivalence relations and fusing them into a single universal postcondition.

We then gain the following benefits:

1. Deferred equivalence checking: programmers can refactor via *sequences* of transformations.
2. Behaviour evolution: users can deliberately alter program behaviour during metamorphosis.
3. Decoupled behavioural equivalence and transformations: we can now safely add new transformations without having to consider behavioural equivalence. Conversely, any refinements we add to behavioural equivalence benefit all transformations.

3 An Example

To see how program metamorphosis might function in practice, consider the following Standard ML [12] code fragment, here presented together with its referencing environments:

Suppose that the programmer wants to swap the names of "result" and "num". The obvious approach of using two *rename* refactorings [6], one to rename "result" to "num" and the other to rename "num" to "result", does not work: Renaming "result" to "num" causes the expression num + 1 to use the wrong "num" due to name capture, and renaming "num" to "result" causes name capture in x + result. In other words, naively composing two rename *refactorings* does not solve the problem, and therefore traditional refactoring (as implemented in e.g. HaRe [9], Eclipse [15], and IntelliJ) is

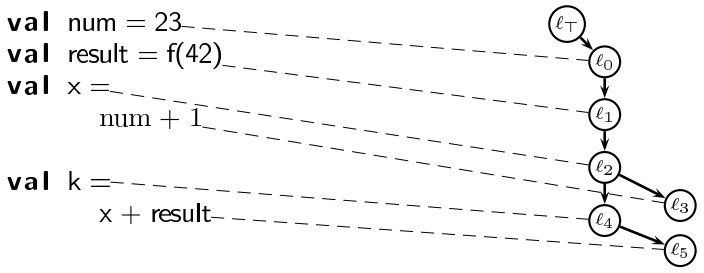


Figure 2: An SML example, together with its scoping structure.

insufficient. This is a straightforward example of a situation where program metamorphosis can help.

Let us try to construct a program metamorphosis system for such name-based transformations: an appropriate metamorphosis system must handle name analysis, usually the first stage of semantic program analysis. Correspondingly, its equivalence relation ($\equiv$) should be defined on the results of name analysis. This system need not support any other kinds of equivalence, so we can require syntactic equivalence for the remaining parts of the program. Then, our metamorphosis system looks as follows:

- properties (the program properties abstraction) represents names by unique identifiers, matching up identifier uses with their corresponding definitions. The properties we need to represent are the following:

  - Name bindings, mapping unique *identifiers* to their names. Here, identifiers are unique objects we introduce for each variable definition. For example,

    ```
    let val v = 0
    in let val v = 1   ...
    ```

    uses only one name, but contains two identifiers. We write $i \overset{n}{\mapsto} n$ to relate identifier i to name n.
  - The *definitions* of identifiers, associating identifiers with the program locations that define them. $\mathsf{Def}(i, \ell)$ indicates that i is defined at ℓ.
  - The *uses* of identifiers, relating identifiers to program locations that use them. $\mathsf{Use}(i, \ell)$ indicates that i is used at ℓ.
  - Scope enclosure, which relates two locations to express the program's scoping structure. We use the binary infix relation $\ell' \prec \ell$

to describe that the location ℓ' is the location of the immediately enclosing scope of location ℓ. Figure 2 shows the scoping structure of our program. The nodes on the right-hand side represent program locations, the dashed lines indicate the connection between the locations and the source program, and the solid arrows between the location nodes represent the relation ($\prec$), i.e., scope enclosure.

- $V_{\mathcal{M}}$ (the validity check) ensures that the program contains no name-based inconsistencies, such as name capture.
- ($\equiv_{\mathcal{M}}$), finally, compares two results of properties to see whether and how their individual names and uses can be matched up, i.e., whether the program structure is isomorphic.

With program metamorphosis, swapping the names of "num" and "result" becomes easier. First, consider how we represent the definitions and uses of "num":

$$\mathsf{Def}(i_{\mathsf{num}}, \ell_0), \mathsf{Use}(i_{\mathsf{num}}, \ell_3)$$

These two directly relate "i_{num}", the identifier we use for the initial "num", to its definition and its one use in Figure 2. The properties for "result" look similar:

$$\mathsf{Use}(i_{\mathsf{result}}, \ell_5), \mathsf{Def}(i_{\mathsf{result}}, \ell_1)$$

Let us first rename "result" to "num", both in its use (line 4 in the ML program and ℓ_5 in the model) and in its definition (line 2 and ℓ_1). As a result, i_{result} at line 2 (ℓ_1) now captures i_{num} from line 1 (ℓ_0), so that the reference to "num" in line 3 (ℓ_3) references the wrong identifier. This yields a single change in the resulting program model: we now have $\mathsf{Use}(i'_{\mathsf{result}}, \ell'_3)$ instead of the $\mathsf{Use}(i'_{\mathsf{num}}, \ell'_3)$ we expected.

But the metamorphosis system lets us continue, so we now rename "num" in lines 1 and 3 to "result". This resolves the name capture; after two steps, we encounter a program model isomorphic to the one we started out with and are done.

4 Abstract Implementation Strategy

Our example illustrated what a program metamorphosis system might look like, but the example exposed two weaknesses in the straightforward implementation:

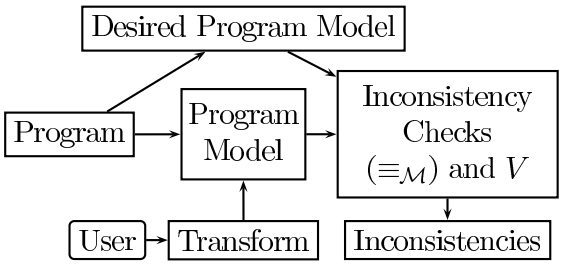


Figure 3: Inconsistency detection process for stateful program metamorphosis.

1. We had to re-compute the program model after each transformation. Apart from being inefficient, this may be hard if the intermediate program is invalid.
2. By recomputing the program model, we could not directly compare information from the initial program with the modified program: in our example, we had to resort to an isomorphism check to indirectly determine that "something" was going wrong. If we had retained information from the initial model, we could have associated location ℓ_3 with the desired use of i_{num}; we could then have established the inconsistency directly and locally by finding that the name of num resolves to i_{result} at ℓ_3 instead.

For the above reasons we find it useful to retain all or part of the program model from previous steps. In the following, we introduce program metamorphosis systems that employ this approach, and give an extended example.

4.1 Stateful Program Metamorphosis

Recall our earlier notion of a program model $M \in \mathcal{M}$: such M describe program semantics by representing all relevant program properties. Our function $\mathsf{properties} : L \to \mathcal{M}$ computes these models via standard program analysis techniques, and $(\equiv_{\mathcal{M}})$ determines whether two elements of $\mathcal{M}$ are equivalent in the sense that they describe programs with the same behaviour. If we could now predict the effect that individual transformations t have on program models from $\mathcal{M}$, we could significantly simplify our equivalence checks: we would only have to reconsider the altered parts of the model when determining validity and equivalence.

We achieve this with an architecture as shown in Figure 3. From the original program, we derive a *desired program model*. This we copy into a *(current) program model*. Whenever the user transforms the program, we simultaneously update the current model. We can then check the current model for validity and for equivalence with the desired model, and report *inconsistencies* whenever they arise.

To transform model and program in parallel, we pair each transformation $t : L \to L$ with a *model transformation* $t_m : \mathcal{M} \to \mathcal{M}$ that captures the effect of t on program models– consequently, we require that $\mathsf{properties} \circ t = t_m \circ \mathsf{properties}$. In practice, we can find representations for M and t_m that allow us to efficiently compute differences between M and $t_m(M)$ (cf. Section 4.3).

Since this implementation strategy allows us to retain much of the previous analysis state, we call it *stateful program metamorphosis*; in the remainder of this document, we focus on this strategy.

Note that the improved efficiency we gain from retaining state comes at a price: we lose the pleasant property that the correctness of a program metamorphosis system is independent of the transformations it provides, and we need to require additional information from transformations (namely model transformations). Both losses are modest when compared to the benefits we retain over traditional refactoring systems.

4.2 Properties of Stateful Program Metamorphosis

On a high level, our stateful strategy seems convenient, but it may not be obvious how we can apply it in practice. What would the state of such a system look like? Can we incorporate all properties we are interested in? And do we have to model the *entire* behaviour of the program for metamorphosis? Before adopting this strategy, we should examine the above questions more closely.

We begin with the last of these questions: stateful program metamorphosis systems need not model the entire behaviour of the program. In fact, we generally assume that stateful metamorphosis systems split the program behaviour into the program model (i.e., $M \in \mathcal{M}$), which they retain, and secondly the rest, which they expect transformations to preserve. Unfortunately, it is not clear what this latter requirement means: let's say that we are transforming one invalid program to another; how can this step possibly "preserve" any behaviour?

We work around this question by giving our programs a *proto-semantics*, as follows:

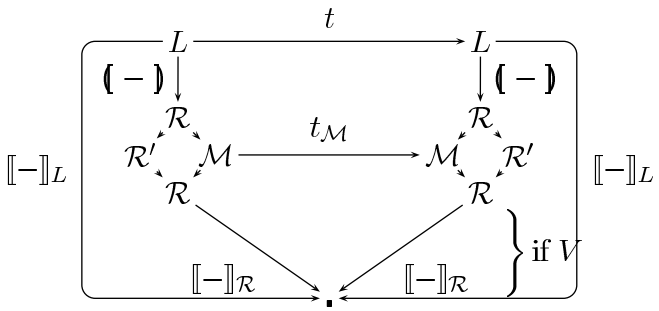


Figure 4: Overview of the construction of stateful program metamorphosis, for Section 4.2. L is the programming language, $\mathcal{R} = \langle \mathcal{M}, \mathcal{R}' \rangle$ is the protosemantics, with $\mathcal{M}$ being the class of program models (we mirror this presentation for the left $\mathcal{R}$). Note that properties (not presented here) is $(\!|-|\!)$ followed by left projection; thus $\mathsf{properties} \circ t = t_{\mathcal{M}} \circ \mathsf{properties}$. $[\![-]\!]_L$ and $[\![-]\!]_{\mathcal{R}}$ are not total.

Definition 3. *Assume a language L with a validity predicate V and semantics $[\![-]\!]_L$. A* proto-semantics *of L is some $\mathcal{R}$ with two mappings $(\!|-|\!) : L \to \mathcal{R}$ and $[\![-]\!]_{\mathcal{R}}$ such that $(\!|p|\!)$ is well-defined for all p and $[\![p]\!]_L = [\![(\!|p|\!)]\!]_{\mathcal{R}}$ whenever $V(p)$.*

(In the following, we drop the subscripts whenever the semantics function is unambiguous.) The intuition behind the above definition is that we have something "less" than a full denotational semantics in the sense that we don't necessarily map programs with identical meaning to the same objects. However, our proto-semantics is defined for *all* programs, irrespectively of their validity, and we require it to respect our "normal" notion of semantic equivalence.

We can now choose our $\mathcal{R}$ as a categorial product, to split between the program model and the rest of the semantics. For simplicity of exposition, we choose a more concrete formalisation:

Definition 4. *Let R be an indexed family of relations R_i with $1 \leq i \leq n$. Let $\mathcal{R}$ be the set of all R, and assume functions $[\![-]\!]_{\mathcal{R}}$ and $(\!|-|\!)$. The triple $\langle \mathcal{R}, [\![-]\!]_{\mathcal{R}}, (\!|-|\!) \rangle$ is a* relational proto-semantics *for language L with semantics $[\![-]\!]_L$ iff it is a proto-semantics of L.*

Such relations may relate identifiers with their names or with their defining locations; they may relate modules with their bodies or their surrounding

modules, types with their supertypes or basic blocks with their dominators. They may even relate noncomputable objects, such as identifiers with their denotational semantics.

Relational proto-semantics are sufficiently expressive for anything we can represent in set theory. They also give rise to a straightforward strategy for splitting "covered" from "non-covered" semantics, which we write $\mathcal{R} = \mathcal{M}, \mathcal{R}'$ where $\mathcal{M}$ is again our program model (the "covered" part) and $\mathcal{R}'$ is the rest. More precisely, we choose (without loss of generality) some k with

$$\begin{aligned} \mathcal{M} &= \{\langle r_1, \ldots, r_k \rangle | \langle r_1, \ldots, r_n \rangle \in \mathcal{R}\} \\ \mathcal{R}' &= \{\langle r_{k+1}, \ldots, r_n \rangle | \langle r_1, \ldots, r_n \rangle \in \mathcal{R}\} \end{aligned}$$

This split is practical:

Theorem 2. *For any computable properties we can find a relational proto-semantics $\mathcal{R}$ that can be split as $\mathcal{R} = \mathcal{M}, \mathcal{R}'$ such that $\mathcal{M}$ contains precisely the computable properties we seek.*

Proof. Relational proto-semantics is sufficiently powerful to encompass all computable properties, so we only need to show that we can find a split that encodes precisely the computable properties we are interested in as part of $\mathcal{M}$. We take an arbitrary split (which trivially exists with $\mathcal{R}' = \mathbf{1}$ and $\mathcal{M}$ encoding the language syntax) and iterate: as long as some relation in $\mathcal{M}$ may contain relations that are not computable or describes properties we are not interested in, we push the relation to $\mathcal{R}'$. Whenever $\mathcal{R}'$ contains a relation R with information we are interested in, we choose an isomorphic $\langle R_M, R_R \rangle \cong R$, where R_M is a computable approximation of the properties we want, and add R_M to $\mathcal{M}$ and R_R to $\mathcal{R}'$. □

In the following, we focus on relational proto-semantics and use the notation $\langle M_p, R_p \rangle = (\!|p|\!)$ to conveniently express the *split proto-semantics* of a program as justified by the above discussion.

Since our proto-semantics provides us with both $\mathcal{M}$ and with the information missing from $\mathcal{M}$, it is natural to wonder how it relates to our equivalence relation ($\equiv_{\mathcal{M}}$) and to properties. We find that the proto-semantics gives us a natural way to define both. properties(p) is simply M_p from $\langle M_p, R_p \rangle = (\!|p|\!)$. ($\equiv_{\mathcal{M}}$) is more tricky. Let us first construct an auxiliary *proto-semantic equivalence relation* ($\equiv_{\mathcal{R}}$) as follows:

$$(\!|p|\!) \equiv_{\mathcal{R}} (\!|p'|\!) \implies V(p) \wedge V(p') \implies [\![p]\!] = [\![p']\!]$$

$(\equiv_{\mathcal{R}})$ is thus an arbitrary conservative approximation of the kernel of $[\![-]\!]$, whenever the programs it considers are valid.

Then, the relation $(\equiv_{\mathcal{M}})$ is a conservative approximation over some $(\equiv_{\mathcal{R}})$, and we can characterise it as being agnostic of the $\mathcal{R}'$ part of the proto-semantics as follows:

Definition 5. *A relation* $(\equiv_{\mathcal{M}})$ approximates *a proto-semantic equivalence relation* $(\equiv_{\mathcal{R}})$ on $\mathcal{M}$ *whenever*

$$p \equiv_{\mathcal{M}} p' \iff \forall x \in \mathcal{R}'. \langle M_p, x\rangle \equiv_{\mathcal{R}} \langle M_{p'}, x\rangle \tag{1}$$

Lemma 1. *Let* $(\equiv_1)$, $(\equiv_2)$ *be proto-semantic equivalence relations. Then there exists a proto-semantic equivalence relation* $(\equiv_{1,2})$ *such that* $(\equiv_i) \subseteq (\equiv_{1,2})$, $i \in \{1,2\}$, *and all* $(\equiv_{\mathcal{M}})$ *that approximate* $(\equiv_1)$ *and* $(\equiv_2)$ *on* $\mathcal{M}$ *approximate* $(\equiv_{1,2})$ *on* $\mathcal{M}$.

Proof. We construct $(\equiv_{1,2}) = (\equiv_1) \cup (\equiv_2)$. □

With this, we are finally ready to define soundness of transformations:

Definition 6. *A* predictive transformation *for a given stateful program metamorphosis system is a tuple* $\langle t, t_m\rangle$ *where* $t : L \to L$ *and* $t_m : \mathcal{M} \to \mathcal{M}$. *A predictive transformation is*

1. consistent *whenever* $\mathsf{properties} \circ t = t_m \circ \mathsf{properties}$
2. covered *whenever, for all* $p \in L$ *with* $\langle M_p, R_p\rangle = (\!|p|\!)$, *there exists some proto-semantic equivalence relation* $(\equiv_{\mathcal{R}})$ *approximated by* $(\equiv_{\mathcal{M}})$ *such that*
$$\forall x. (\!|\langle M_p, x\rangle|\!) \equiv_{\mathcal{R}} (\!|\langle t_m(M_p), x\rangle|\!) \implies (\!|p|\!) \equiv_{\mathcal{R}} (\!|t(p)|\!)$$
3. fully covered *whenever* $(\!|t|\!) = \langle t_m, \mathsf{id}\rangle$
4. sound *whenever it is consistent and covered.*

Lemma 2. *Any fully covered transformation is sound.*

We can now show the correctness of stateful metamorphosis:

Definition 7. *A* stateful (program) metamorphosis system *is a program metamorphosis system with* $\mathsf{properties}$ *and with an accompanying relational proto-semantics.*

Theorem 3. *A stateful program metamorphosis system guarantees the preservation of behaviour if all of its transformations are sound. Specifically, for any sequence of transformations* $\overline{t} = t_n \circ \cdots \circ t_1$,

$$V(p) \wedge V(\overline{t}(p)) \implies (\!|p|\!) \equiv_{\mathcal{M}} (\!|\overline{p}(p)|\!) \implies [\![p]\!] = [\![\overline{t}(p)]\!]$$

Proof. We show the above property indirectly, by proving that $(\!|p|\!) \equiv_{\mathcal{M}} (\!|\overline{t}(p)|\!) \implies (\!|p|\!) \equiv_{\mathcal{R}} (\!|\overline{t}(p)|\!)$ (which implies the desired property). In our proof, we encounter multiple proto-semantic equivalence relations; by Lemma 1, this is not a problem. *Induction anchor*: Here, $\overline{t} = \mathsf{id}$; this case is trivial. *Induction step*: Assume $(\!|p|\!) \equiv_{\mathcal{M}} (\!|\overline{t}(p)|\!) \implies (\!|p|\!) \equiv_{\mathcal{R}} (\!|\overline{t}(p)|\!)$. Let t' be a sound predictive transformation. To show $(\!|p|\!) \equiv_{\mathcal{M}} (\!|t' \circ \overline{p}(p)|\!) \implies (\!|p|\!) \equiv_{\mathcal{R}} (\!|t' \circ \overline{t}(p)|\!)$ we consider two possibilities: if $(\!|p|\!) \equiv_{\mathcal{M}} (\!|t' \circ \overline{p}(p)|\!)$ does *not* hold, we are done. Otherwise this formula entails the premise of the coveredness property of t', which then implies $(\!|p|\!) \equiv_{\mathcal{R}} (\!|t' \circ \overline{t}(p)|\!)$. □

We did not need the consistency property of our transformations for the above proof, since we only require this property to speed up implementations of stateful program metamorphosis.

The above results tell us that we can use stateful program metamorphosis to model all interesting program properties, as long as they are computable, and they give us a meta-theory for proving the soundness of transformations in such a system. In particular, we find that transformations that are fully characterised by their model transformations are implicitly sound (Lemma 2).

4.2.1 Explicit Behavioural Changes

In addition to the properties we discussed above, stateful metamorphosis gives rise to a convenient means for *evolving* behaviour, one of the central premises of program metamorphosis.

Consider two Java classes

```
class A {
    public static Object mkA() { ... }
}
class B {
    Object o = A.mkA();
}
```

If A is part of a library, the refactoring developer might not be aware of class B. Let us now assume that the refactoring developer renames mkA to

genA: this will break the external code in class B, and should therefore be considered a behavioural change.

The above renaming is an example of renaming publicly visible methods, which in turn is an example of a transformation with potential behavioural change.

Whenever the current program model disagrees with the desired program model about the library exports, we can give users the option to update the desired program model to indicate agreement with the observed change.

4.3 Ad-Hoc Implementation of Stateful Program Metamorphosis

Stateful program metamorphosis can be implemented in many ways; one straightforward approach is to manually implement program analyses and represent analysis results in a way that is optimised for quick inconsistency detection. We refer to this approach as "ad-hoc implementation", since it requires no formal structure.

To give an example of the ad-hoc approach, consider the implementation of one of our two prototypes for SML refactoring (cf. Section 5): This system is based on a program dependence graph [5] that we construct from the program (similarly to traditional refactoring approaches such as Griswold's [7]). We annotate program locations with all relevant properties, including

- The complete local referencing environment, represented as a splay tree [16] based map.
- A set of exported identifiers and names, for each module and for the toplevel declaration sequence.

Storing the local referencing environment helps us to perform partial updates: whenever a property changes due to a transformation, we update only the affected subgraphs of the program, though further bookkeeping is required to properly maintain the list of inconsistencies.

4.4 Stateful Program Metamorphosis in Datalog

Another possible strategy for implementing program metamorphosis is to use Datalog [18, 20]. Our first prototype was based on this approach, but we found it necessary to abandon its implementation due to significant performance problems.

Nonetheless, Datalog offers two advantages over the ad-hoc implementation approach: First, Datalog can be formalised easily (modulo negation/stratification) and therefore easily described in terms of relational algebra, and secondly, it is sufficiently simple to give rise to heuristics for recovery planning. We do not explore the latter in this document, though we exploit the former. To simplify our exposition, we will utilise a Datalog-style relational notation, explaining unusual constructs as we encounter them.

5 Implementing Stateful Program Metamorphosis for SML

To evaluate our approach, we implemented two prototypes of a refactoring system based on stateful program metamorphosis, one using Datalog and another based on an ad-hoc strategy. Both use the SML of New Jersey frontend [1] for semantic analysis. The implementations represents SML programs as values of an algebraic datatype and perform AST transformations and model updates in parallel. Both prototypes cover the same functionality.

In the following, we describe the concrete program metamorphosis system we use in Datalog-style relational terms. Due to space restrictions, we limit our discussion to the SML subset "Small ML", introduced in the next section, and informally describe how we support some of the remaining SML features in Section 5.2.

5.1 Small ML

Syntactically, Small 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*.

Small 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 Small ML are either directly applicable or extend straightforwardly to the entirety of Standard ML, as discussed in Section 5.2.

We choose Small ML over other languages such as Mini-ML [4] since most "restricted ML" variants focus on the dynamic semantics of ML, whereas our needs are largely due to syntax and static semantics.

In the following, we describe the components of a Datalog-based stateful metamorphosis system for Small ML. Section 5.1.1 describes the ground model we consider, Section 5.1.2 details the program theory, and Section 5.1.3 lists our transformations.

5.1.1 Small ML Program Models

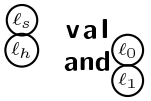
We use the following sorts in our system: NAME (the sort of names), ID (the sort of identifiers), and LOC (the sort of locations).

To support nominal and structural transformations in Small ML, we require a number of properties expressed as ground relations. Table 1 summarises these relations. Some of the relations we already used in Section 3; we repeat them for completeness. Below, we discuss some of the more involved relations in detail. We use the notation

$$R : S_1 \times \ldots \times S_n$$

to mean that $R \subseteq S_1 \times \ldots \times S_n$ is a relation between the sorts $S_1 \ldots S_n$.

- $\mathsf{PDef} : \text{ID} \times \text{LOC}$. Records definitions of identifiers as parameters. Such definitions are not visible to subsequent definitions, only to the body of the definition (transitively reachable via $\prec$).
- $\mathsf{Def} : \text{ID} \times \text{LOC}$ and $(\lhd) : \text{LOC} \times \text{LOC}$. We use Def in a slightly different sense as in Section 3: $(\lhd)$ and Def together represent the definitions of identifiers with nontrivial scoping rules. Definitions made via $\mathsf{Def}(-, \ell)$ and $\ell_h \lhd \ell$ 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:

  $i_0 = 7$
  $i_1 = 13$

  We observe the following facts:

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

  To complete the above **val** definition, we add facts $\ell_s \prec \ell_0$ and $\ell_s \prec \ell_1$. The bodies of i_0 and i_1 now draw their environments from 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 ℓ_h via $(\prec)$, and therefore (via $\ell_h \triangleleft \ell_0$ and $\ell_h \triangleleft \ell_1$) 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 $(\triangleleft)$ 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.

- $(\triangleleft)$: Loc×Loc, Head : Loc, ValHead : Loc and Val : Loc. These four relations structure value definition blocks. Recall that each **val** definition block may define multiple identifiers, combined by the keyword **and**. We associate precisely one location ℓ_h with each such definition block; we call ℓ_h the *definition head* and mark it with ValHead and Head. For each defined identifier, we introduce a location ℓ_b which we call the *definition branch* and mark as Val. We then relate each definition branch ℓ_b with its unique associated definition head ℓ_h via $\ell_h \triangleleft \ell_b$, such that each head may have many branches, but each branch has precisely one head.

- $(\triangleleft)$: Loc × Loc, Head : Loc, FunHead : Loc and Fun : Loc. These four relations structure function definition blocks, analogously to value definition blocks (except that FunHead is used instead of ValHead and Fun instead of Val). We mark both kinds of definition heads as Head but can distinguish them by observing whether they are in FunHead or ValHead.

5.1.2 Analyses and Inconsistencies for Small ML

The following conditions render Small ML programs invalid, in terms of our V predicate:

- Ambiguous(ℓ, 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 [12] dictates this condition.

- VarCapture(i, ℓ_u, ℓ_c): the use of the identifier i at location ℓ_u is subject to name capture at location ℓ_c

Relation	Meaning
$\ell \prec \ell'$	Location ℓ immediately precedes ℓ' in scope
$i \overset{n}{\mapsto} n$	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
$\ell_h \lhd \ell_b$	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 1: Ground relations for Small ML

- $\mathsf{VarNonReach}(i, \ell_u)$: 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 5 specifies the rules for deriving these conditions.
We use three auxiliary relations in this figure:

- $(\ll) : \textsc{Loc} \times \textsc{Loc}$, which is the transitive closure of $(\prec)$.
- $\Delta : \textsc{Id} \times \textsc{Loc}$, where $\Delta(i, \ell)$ states that identifier i is defined at location ℓ, i.e., $\Delta = (\lhd \circ \mathsf{Def}) \cup \mathsf{PDef}$.
- $\mathsf{Reach} : \textsc{Id} \times \textsc{Loc}$, 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 5, we see the inference rules for the various error conditions. Ignoring (for the time being) the use of $\mathsf{InLimbo}$, 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

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

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

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

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

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

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

Figure 5: Datalog rules for deriving error conditions for Small ML, written in Natural Deduction style

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. VarNonReach(i, ℓ_u) holds iff we use i at location ℓ_u, but cannot actually reach it from there.

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

5.1.3 Transformations for Small ML

Figure 6 lists a number of transformations for Small ML. These transformations subsume a number of standard refactorings, specifically identifier renaming and definition relocation (Section 6.1). 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., Def(i, ℓ_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 5. 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
```

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

$$\textsf{elim-def}(i,\ell_b) = \quad \frac{\textsf{Def}(i,\ell_b) \quad \ell_h \lhd \ell_b \quad \ell_h \prec \ell_b}{\textsf{Def}(i,\ell_b) \quad \textsf{Limbo}(\ell_b)}$$

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

$$\textsf{intro-fun}(\ell_b,\ell_h) = \quad \frac{\textsf{Fun}(\ell_b) \quad \textsf{Limbo}(\ell_b) \quad \textsf{FunHead}(\ell_h)}{\textsf{Fun}(\ell_b) \quad \textsf{FunHead}(\ell_h) \quad \ell_h \lhd \ell_b \quad \ell_h \prec \ell_b}$$

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

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

Figure 6: Transformations for Small ML

```
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*, exception for function definition heads.

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 5); 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.

5.2 Modelling Standard ML

In addition to the transformations we discussed for Small ML, our prototypes include the behaviour-changing extension listed in Section 4.2.1, by providing transformations to explicitly accept the elimination and renaming of previously exported identifiers from the toplevel and from structures, as well as a transformation to explicitly accept the introduction of a new identifier.

Our prototypes further add type aliases, and allow substituting type identifiers by aliases.

Our prototype implementations handle most of Standard ML; the aspects of SML it is missing are abstypes, signatures, functors, fixity declarations, and side effects in val bindings.

5.3 Soundness

Our implementation is not (yet) sound for all of Standard ML, since some of the transformations do not properly model side effects. To the best of our knowledge, this is a limitation we share with all existing refactoring systems for impure languages. We expect to add support for side effects by introducing a conservative effect analysis into our program model and allowing behaviour-evolving transformations (Section 4.2.1) to allow users to override the conservative results from this analysis.

6 Evaluation of our Prototype

We evaluated our prototype implementations in two ways: first, by comparing the transformations they support to those listed in a standard catalogue of refactorings (Section 6.1), and secondly by experimenting with the planner, to generate recovery plans (Section 6.2). The first evaluation is identical for both prototypes. The second varies considerably; we only report numbers for our ad-hoc implementation, since the runtime performance of our Datalog prototype renders it impractical.

6.1 Supported refactorings

Presently, Thompson and Reinke's Catalogue of Functional Refactorings [17] 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 mechanisms, and #18 due to the lack of syntactic sugar for Monads). We implemented the transformations listed in Figure 6, transformations for new type aliases and 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.

Program	Init (best)	Inconsistency (best)
sample	$42\mu s$	$22\mu s$
life	$182\mu s$	$59\mu s$
ray	$571\mu s$	$112\mu s$

Figure 7: Performance of program metamorphosis. "Init" refers to the time needed to abstract the desired program model from ground facts. "Inconsistency" is worst-case (whole-program) inconsistency inference. "Update" is the average time needed to recompute inconsistencies after a transformation.

6.2 Initial Results

We evaluated our implementation by applying it to three programs, the four-line sample program we used as an example in Section 3, and two programs from the MLton[1] 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.

Figure 7 summarises our results. We ran our implementation (compiled by the MLton whole-program optimising compiler, revision 20070826), on a 2.16 GHz Core2Duo machine. For our performance results, we reported the best number out of 100 runs.

As we see, 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. We therefore expect program metamorphosis to be practical even for very large programs.

6.3 Experiences with Recovery Plan Search

In our experiments, we found the inconsistencies reported by our system to be easy to trace, allowing us to quickly restore programs to validity. For larger programs and for novice users, automatic support might be helpful. Since program metamorphosis gives rise to a notion of automatic recovery planning, i.e., searching for sequences of transformations that restore programs to validity, we added such search facilities to our prototypes.

We implemented two planners: one "intelligent" planner with search heuristics, for our Datalog system, and one "dumb" planner, performing

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

greedy search, for our ad-hoc system. While both systems are currently not sufficiently efficient for practical use, they report promising results: for our earlier example (Section 3), they 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 former "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. We expect that combining the efficient search space pruning of our Datalog system with the fast evaluation times of our ad-hoc planner will yield a planner that is practical at least for small and medium-sized programs.

7 Discussion

We have presented program metamorphosis as an alternative to traditional machine-supported refactoring. In the following, we compare our approach and traditional refactoring in several additional respects: the user interface (Section 7.1), scalability, (Section 7.2), and language specificity (Section 7.3).

7.1 User interface

Since program metamorphosis normally operates on an *abstracted* representation of the program under metamorphosis, it must not allow free-form editing while metamorphosis is in progress. Thus, editors with program metamorphosis support must be bimodal; for example, our prototype implementation utilises two different EMACS major modes to achieve this effect. Users first switch into "program metamorphosis mode" before applying transformations, and switch back once they are done. During metamorphosis, the editor must visualise inconsistencies.

Traditional refactoring does not require modality. However, practical refactoring tools (such as used in Eclipse) often already include a preview

mechanism and other complex UI elements, many of which can be avoided since program metamorphosis requires no previewing and can employ more fine-grained transformations.

7.2 Scalability

Program metamorphosis requires both a current and a desired program model. Thus, program metamorphosis must, in theory, represent up to twice as much data as a traditional refactoring system.

While we cannot rule out the existence of practical cases in which program metamorphosis indeed suffers from this overhead, our experiences indicate that this discrepancy is often less severe:

- Oftentimes, the desired program model needs only an abstraction of the full program model. For example, in our prototype the desired program model only consists of the set of currently used identifiers, at each program location, and the set of exported identifiers plus expected names, at each module definition and at the toplevel.
- Existing refactoring systems often also represent such abstractions, to facilitate operations not related to refactorings (such as jumping to the definition of a selected identifier).
- Computing the precondition of a refactoring necessarily includes overhead of its own. In general, preconditions may need to simulate a full program transformation: then, their overhead is no less than ours.

7.3 Language Specificity

Refactoring has been applied to object-oriented [15], functional [9], and logic programming languages [14]. By contrast, we have only discussed program metamorphosis in Standard ML. To illustrate that our approach is not specific to any particular language paradigm, we have also begun implementing a program metamorphosis system for Java, using an ad-hoc implementation approach (Section 4.3) and exploiting the RECODER library[2]. This system is not yet complete, but our current observations indicate that at least structural and nominal transformations in Java are no harder than in functional languages; for example, inherited properties in Java behave very similarly to opened structures in SML.

[2] `http://recoder.sourceforge.net`

8 Related Work

There is a large body of related work on refactoring (cf. [11] for a survey), including many implementations, such as HaRe [9] and Eclipse [15]. The observation that more information than immediately visible to the eye is needed to perform correct transformations was already employed by Griswold [7], who used Program Dependence Graphs [5] for this purpose. These systems consider refactorings to be individual macroscopic transformations. Some other program transformation approaches [2, 19] look specifically for atomic transformations, but remain entirely semantics-preserving.

Composing transformations to achieve a certain goal is the central theme of AI Planning (cf. [13] for a high-level overview). The composition of refactorings in particular has also been considered [8], but only for traditional approaches to refactoring, without allowing intermediate invalidation of correctness properties.

9 Conclusion

We have presented a novel approach to program refactoring that defers correctness checks until after a transformation sequence of arbitrary length. Our approach permits such sequences to be constructed interactively, maximising user control. Simultaneously, our approach permits judicious behaviour evolution. Since the scope of our approach is greater than that of refactoring, we distinguish it from the latter by referring to ours as *program metamorphosis*.

We refined program metamorphosis it into *stateful program metamorphosis*, which retains prior program state to speed up correctness checks. Our experimental results suggest that stateful program metamorphosis is practical, can be implemented efficiently, and constitutes a viable extension over traditional refactoring.

Acknowledgements

The authors are indebted to Daniel von Dincklage, Devin Coughlin, William Griswold, Jeremy Siek, Philipp Wetzler, and the anonymous POPL and ICFP referees for their valuable feedback on this work.

References

[1] Andrew W. Appel and David B. MacQueen. A Standard ML Compiler. In *Proc. of the Conf. on Functional Prog. Lang. and Comp. Arch.*, volume 274, pages 301–324, Portland, OR, 1987. Springer, Berlin.

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

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

[4] 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.

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[6] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] W. G. Griswold and D. Notkin. Program Restructuring as an Aid to Software Maintenance. Technical report, Univ. of Wash., 1990.

[8] Günter Kniesel and Helge Koch. Static composition of refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, 2004.

[9] Huiqing Li, Claus Reinke, and Simon Thompson. Tool Support for Refactoring Functional Programs. In J. Jeuring, editor, *ACM Sigplan Haskell Workshop*, pages 27–38, 2003.

[10] 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.

[11] Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[12] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML – Revised*. MIT Press, Cambridge, MA.

[13] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

[14] T. Schrijvers, A. Serebrenik, and B. Demoen. Refactoring prolog programs, 2001.

[15] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developers Guide to Eclipse*. Addison-Wesley, May 2003.

[16] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[17] Simon Thompson and Claus Reinke. A Catalogue of Functional Refactorings, Version 1, 2001.

[18] 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.

[19] Martin P. Ward and Hussein Zedan. MetaWSL and Meta-Transformations in the FermaT Transformation System. In *COMPSAC (1)*, pages 233–238, 2005.

[20] 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. Springer-Verlag, November 2005.