Specifying and Executing Optimizations for Generalized Control Flow Graphs

William Mansky\textsuperscript{a}, Elsa L. Gunter\textsuperscript{b}, Dennis Griffith\textsuperscript{b}, Michael D. Adams\textsuperscript{c}

\textsuperscript{a}Department of Computer and Information Science, University of Pennsylvania, Levine Hall, 3330 Walnut Street, Philadelphia, PA 19104-6309
\textsuperscript{b}Department of Computer Science, University of Illinois at Urbana-Champaign, Thomas M. Siebel Center, 201 N. Goodwin, Urbana, IL 61801-2302
\textsuperscript{c}School of Computing, University of Utah, 50 S. Central Campus Drive, Room 3190, Salt Lake City, UT 84112

Abstract

Compiler optimizations, usually expressed as rewrites on program graphs, are a core part of modern compilers. However, even production compilers have bugs, and these bugs are difficult to detect and resolve. In this paper we present Morpheus, a domain-specific language for formal specification of program transformations, and describe its executable semantics. The fundamental approach of Morpheus is to describe program transformations as rewrites on control flow graphs with temporal logic side conditions. The syntax of Morpheus allows cleaner, more comprehensible specifications of program optimizations; its executable semantics allows these specifications to act as prototypes for the optimizations themselves, so that candidate optimizations can be tested and refined before going on to include them in a compiler. We demonstrate the use of Morpheus to state, test, and refine the specification of a variety of transformations including a simple loop peeling for single-threaded code and a redundant store elimination optimization on parallel programs.

Keywords: optimizing compilers, program transformations, temporal logic, SMT solvers

1. Introduction

Of the various phases of a modern compiler, optimization is generally considered to be the most complex. At the point of optimization, programs have
usually been parsed and transformed into some internal representation—most often a control flow graph in which nodes are labeled with instructions in some intermediate language and edges represent jumps in control flow. Before generating the low-level code that actually executes on a machine, the compiler attempts to rearrange the graph to improve its time and memory performance without introducing new behavior to the program. While the transformations involved may be simple, the conditions under which they are safe to apply, which often rely on extensive program analysis, are easily misstated. In practice, even widely used compilers such as GCC have been shown to transform code incorrectly [1]. Compiler designers stand to benefit from techniques for specifying optimizations more systematically.

In this article, we present a domain-specific language for specifying compiler optimizations and transformations, called Morpheus. The language follows the work of Lacey et al. [2] in looking at optimizations as rewrites on control flow graphs with temporal logic side conditions. Temporal logic formulae over program graphs allow us to simply and clearly state the conditions under which an optimization should be applied. We diverge from previous graph rewriting languages by focusing on transformations that preserve the well-formedness of control flow graphs, allowing us to decompose complex transformations into smaller pieces that are each individually safe to perform. Morpheus has both abstract mathematical semantics, derived from its predecessor language TRANS [3], and executable semantics that can be used by compiler designers to test and refine their transformations on actual program graphs. Morpheus is additionally differentiated from TRANS by the introduction of a strategy language with explicit scope and the structure of a Kleene algebra, allowing users to reason about their transformations utilizing their intuition from regular expressions. In this paper, we present the syntax and both the abstract and executable semantics of Morpheus, and illustrate how it can be used to rapidly prototype and test compiler optimizations. Ultimately, we hope that the approach outlined in this paper will assist compiler writers in creating complex, reliable optimizations.

2. The Morpheus Specification Language

The basic approach of the Morpheus specification language is modeled after the TRANS language of Kalvala et al. [3]: optimizations are specified as conditional compositions of rewrites on a generalized control flow graph (GCFG) containing the program’s code. The language is partitioned
into three largely independent components: core graph transformations, conditions given in a variant of Computation Tree Logic (CTL), and a strategy language for building complex transformations out of component transformations and conditions. Intuitively, the rewrite portion of an optimization expresses the local transformation to be made, the condition characterizes the situations in which the optimization should be applied, and the strategy language allows us to build whole-system transformations out of collections of local ones. Morpheus is a special-purpose language for the transformation of GCFGs, and as such is parametrized by aspects of GCFGs, namely node names, node labels (program instructions), and edge labels (marking control flow). Transformation specifications may mention aspects of GCFGs concretely, but more generally, they use pattern variables that will be instantiated with control flow graph components in each specific application. We will use the term “expressions” to refer to patterns built from both concrete entities and metavariables (which will be instantiated with concrete entities when the transformation is applied). We use the term metavariable to refer to the variables in the patterns and expressions in Morpheus transformations, as opposed to the concrete programming variables that will be found in instructions. From our perspective, actual program variables are concrete objects, while the metavariables are acting as mathematical variables. In what follows, we will assume that \( n, m, n_j, m_j \) range over node name expressions, \( \ell, \ell_j \) range over instruction expressions, and \( d, d_j \) range over edge label expressions. The syntax of Morpheus is given by the following grammar:

\[
A ::= \text{add\_node}(n, \ell, (d_1, m_1), \ldots, (d_k, m_k)) \mid \text{remove\_node}(n) \\
\mid \text{relabel\_node}(n, \ell) \mid \text{move\_edge}((n, d_1, m_1), (n, d_2, m_2)) \\
\varphi ::= \text{true} \mid p(\vec{x}) \mid \varphi \land \varphi \mid \neg \varphi \mid \exists x. \varphi \mid AX \varphi \mid AY \varphi \\
\mid A \varphi U \varphi \mid E \varphi U \varphi \mid A \varphi S \varphi \mid E \varphi S \varphi \\
T ::= A \mid \text{Satisfied\_at} n \varphi \mid \text{NOT} T \mid T \setminus T \mid \text{Exists} x. T \\
\mid T \setminus T \mid T ; T \mid T^* 
\]

The syntax of Morpheus consists of actions, conditions, and transformations. The atomic actions \( A \) begin with \text{add\_node} and \text{remove\_node}, which add and remove nodes that have no incoming edges. In the case of \text{add\_node}, the addition only takes place if the node description is well-formed as a GCFG node (i.e., it has the right number and kind of outgoing edges for its instruction label). The \text{relabel\_node} action relabels an existing node with a new instruction, as long as that new instruction is compatible with the existing
edge structure. The only action that operates directly on edges (rather than nodes) is \texttt{move_edge}, which moves the destination of an edge from one node in the graph to another. This set of basic actions is designed to preserve the well-formedness of a GCFG, and it is sufficient to incrementally transform any GCFG into any other. The arguments to the atomic actions represent nodes, instructions and edge labels in the program graph, but may contain \textit{metavariables} that are instantiated to program objects when the rewrites are applied.

Note that the action \texttt{relabel_node} is almost redundant: we could also relabel a node by creating a new node with the new label and with the same outgoing edges as the original node, and then moving each edge that pointed to the original node to point to the new node instead, and finally removing the original node. This is not quite the same as relabeling, however, since we are forced to introduce a new node, while relabeling allows us to keep the original node name.

The \textit{conditions} $\varphi$ of Morpheus are based on First-Order CTL (FOCTL). Starting from a set of atomic predicates $p$, they include all of the usual propositional and temporal operators. We define the additional propositional connectives $\lor$ and $\Rightarrow$ in the standard way. The $S$ ("since") and $Y$ ("yesterday") operators are the past-time counterparts to the $U$ ("until") and $X$ ("next") operators respectively; for instance, $E \varphi_1 S \varphi_2$ holds when there exists some path backwards through the graph such that $\varphi_1$ holds until a previous point at which $\varphi_2$ holds. The derived “finally” and “globally” operators $EF, AF, EG, AG$ are defined from the $U$ operators in the usual way, and their past-time counterparts $EP, AP, EH, AH$ ("at some previous point in some/all paths" and “at all points in some/all paths”) are defined analogously. The existential quantifier $\exists$ is used to quantify over metavariables in a formula: these metavariables may then appear in the atomic predicates of a formula, enhancing the expressive power of the conditions.

At the top level, a transformation $T$ combines conditions and rewrites using \textit{strategies}. The simplest strategy is just to perform an action $A$. The strategy \texttt{SATISFIED_AT} $n \varphi$ acts as the identity transformation if $\varphi$ holds of the GCFG at the node $n$, and returns the empty set if $\varphi$ fails to hold on $n$. Thus \texttt{SATISFIED_AT} $n \varphi$ acts as filter, allowing through only those GCFGs and nodes $n$ that satisfy $\varphi$. On the other hand, \texttt{NOT} $T$ and $T_1 \setminus T_2$ allow us to deselect graphs by the ability to perform a transformation. The transformation \texttt{NOT} $T$ selects those graphs that $T$ cannot transform, i.e., those not in the domain of $T$, and deselects those that it can transform.
The transformation $T_1 \setminus T_2$ restricts the output of $T_1$ to those graphs that could not be produced by $T_2$, i.e., those not in the image of $T_2$. Note the difference between these two filters: NOT $T$ filters based on the complement of the domain of a transformation, while $T_1 \setminus T_2$ filters based on the complement of the range of a transformation. The strategy EXISTS $x.T$ binds $x$ in $T$, limiting its scope to the free occurrences of $x$ in the conditions and actions of $T$. We will abbreviate EXISTS $x$. EXISTS $y$. . . . EXISTS $z.T$ by EXISTS $x y . . . z.T$. Finally, the constructs + and ; allow for choice between and sequencing of two transformations respectively, and the iteration operator $*$ allows for the repeated application of a transformation any number of times.

For a simple example of a Morpheus transformation, assume we have a language of instructions that supports assignments and binary arithmetic expressions. In this setting, if we have a variable assigned the result of applying an arithmetic operation to two constants, we would like to replace the operation with its result. This can be done by the following transformation:

$$\text{simple_constant_folding} \triangleq$$
$$\exists x \ a \ b \ c \ oper.$$
$$\text{Satisfied}_\text{AT} \ n \ \text{stmt}(x = oper(a, b)) \land \text{is const}(a) \land \text{is const}(b) \land$$
$$\text{is const}(c) \land \text{is}(c, oper(a, b)) ;$$
$$\text{relabel}_\text{node}(n, x = c)$$

This is an existentially bound sequence of a condition and an action. (Note that oper is a metavariable that will be bound to an arithmetic operator appearing in the program syntax.) We may apply simple_constant_folding to a program with a node labeled $\text{diff} = 10 - 2$, and the transformation will match $n$ to (the name for) this node, $x$ to $\text{diff}$, $a$ to $10$, $b$ to $2$, oper to $(\text{op -})$, and $c$ to $8$ (because of the clause is($c, oper(a, b)$)), and relabel the node to $\text{diff} = 8$.

Examples of the use of these strategies can be seen in Section 6.1.

3. The Semantics of Morpheus

In this section, we present the semantics of Morpheus. We give semantics for each of the layers of syntax: atomic actions transform graphs, conditions are satisfied by substitutions, and strategies combine actions and conditions into graph transformations.
3.1. Generalized Control Flow Graphs

Since Morpheus is a language for transforming GCFGs, we begin our semantics with the definition of a GCFG. A GCFG is a labeled directed graph in which the number and type of outgoing edges from each node are governed by the instruction label on that node.

Definition 1. Let $\mathcal{I}$ be a target language providing a set of program instructions, and $\mathcal{D}$ be a set of edge labels (directions). An edge typing $T : \mathcal{I} \rightarrow \mathcal{P}(\mathcal{D} \rightarrow \mathbb{N})$ is a function that takes an instruction $instr$ and returns the set of allowed multisets of edge labels that can occur as the edge labels of the outgoing edges from a node labeled with $instr$.

We then define GCFGs as follows:

Definition 2. Given a set of instructions $\mathcal{I}$, a set of edge labels $\mathcal{D}$, and an edge typing function $T$, a generalized control flow graph (GCFG) is a tuple $(N, L, S, E)$ describing a labeled directed graph, where

- $N$ is a finite set of nodes,
- $L : N \rightarrow \mathcal{I}$ is a labeling of nodes with program instructions,
- $S \subseteq N$ is a nonempty set of start nodes for the graph,
- $E \subseteq \mathcal{P}(N \times \mathcal{D} \times N)$ is a set of labeled edges with the restriction that for each $n \in N$, there exists $f \in T(L(n))$ such that, for each direction $d \in \mathcal{D}$, we have $|\{m \in N \mid (n, d, m) \in E\}| = f(d)$. (Note that the type of $E$ implies that there is at most one edge with a given source, target, and label.)

We will refer to the function $f$ as the edge type of the node $n$.

It is worth noting a couple of differences between generalized control flow graphs as defined here and control flow graphs as commonly used in compiler theory. Firstly, we do not incorporate a notion of basic blocks in our definition of generalized control flow graphs. In general, the concept of basic block can be treated as a predicate on a finite sequence of nodes in a generalized control flow graph. Moreover, properties like being the start, the end, or in the middle of a maximal basic block can be expressed in the condition language of Morpheus. Secondly, we allow multiple start nodes, and make no commitment to the existence of exit nodes. Thirdly, we allow
self-loops in our graphs. Depending on the language of instruction labels, a self-loop may be reasonable control flow; for instance, if nodes are labeled with complex programs, a self-loop may just be an indication of recursion in that node.

3.2. The Semantics of Actions

The semantics of actions is given by a partial function \( [A](\sigma, G) \) that takes an action, a substitution function, and a GCFG and returns the GCFG, if any, that results when the action is performed. A substitution function is a function mapping metavariables to objects, where the type of objects is the disjoint union of nodes, instructions, any subcomponents specific to the instruction language as supplied by the user, edge labels, edges, lists of in edges, lists of out edges, and anything else over which we allow metavariables to range. For example, objects include all things that can be the arguments to the interpretation of the atomic predicates. In the definition of the semantics of Morpheus in general, and actions in particular, we will use a standard abuse of notation in applying the substitution to arbitrary subexpressions of Morpheus expressions (formulae, actions, arguments to predicates, arguments to actions, subcomponents of instructions, and so forth) to mean the act of walking through the subexpression and replacing each metavariable by the object given for it by the substitution, provided the object is of the right type to be used in the given context (if an object is of the wrong type for its context, substitution fails and the semantics of the action is undefined). More generally, each action always preserves the well-formedness of the GCFG, if it returns anything.

**Definition 3.** Let \( G = (N, L, S, E) \) be a GCFG over instruction set \( \mathcal{I} \), edge labels \( \mathcal{D} \) and edge typing \( \mathcal{T} \). Let \( \sigma \) be a substitution mapping metavariables to components in \( G \) (or fresh nodes not in \( G \)). Then the semantics of actions is defined as follows:
Notice that the semantics of each action preserves the properties required of a GCFG. Also notice that the semantics of each action is a simple graph operation that can readily be implemented in any common programming language, as long as we have a mechanism for generating fresh nodes for add_node.

3.3. Semantics of Morpheus Conditions

To give semantics to temporal logic formulae over GCFGs, we must define a notion of paths through a GCFG.
Definition 4. Let $G = (N, L, S, E)$ be a generalized control flow graph. Let $\lambda$ be an infinite sequence of nodes in $G$, and $\lambda_i$ denote the $i$-th element of $\lambda$. A forward path from a node $n$ in $G$ is a member of the set of sequences defined by $\text{Paths}(G, n) = \{ \lambda \mid \lambda_0 = n \land (\forall i. \exists d. (\lambda_i, d, \lambda_{i+1}) \in E) \}$. A reverse path from $n$ in $G$ is a member of the set $\text{RPaths}(G, n) = \{ \lambda \mid \lambda_0 = n \land (\forall i. \exists d. (\lambda_{i+1}, d, \lambda_i) \in E) \land (\forall k' \geq k. \lambda_{k'} = \lambda_k) \}$.

The conditions of Morpheus are given in the branching-time temporal logic FOCTL. A CTL formula expresses a property over a (generally infinite) tree of states. In our case, the trees are derived by “unfolding” a generalized control flow graph into paths, as described above. The conditions are first-order in that variables may appear in the atomic state predicates $p$, and we can quantify over these variables with the $\exists$ operator. The semantics of an FOCTL formula is given by a satisfaction relation as follows:

Definition 5. Let $G$ be a GCFG, $\sigma$ be a substitution of values for metavariables, $n$ be a node in $G$, and $\varphi$ be a FOCTL formula. The satisfaction of $\varphi$,
written $G, \sigma, n \models \varphi$, is defined by structural recursion on formulae as follows:

$G, \sigma, n \models \text{true}$

$G, \sigma, n \models p(\vec{x})$ iff $p(\sigma(\vec{x}))$ is true at $n$ in the semantics for $p$ provided by the target language

$G, \sigma, n \models \varphi_1 \land \varphi_2$ iff $G, \sigma, n \models \varphi_1$ and $G, \sigma, n \models \varphi_2$

$G, \sigma, n \models \neg \varphi$ iff $G, \sigma, n \not\models \varphi$

$G, \sigma, n \models \forall \lambda \in \text{Paths}(G, n). G, \sigma, \lambda_1 \models \varphi$

$G, \sigma, n \models \forall \lambda \in \text{RPaths}(G, n). G, \sigma, \lambda_1 \models \varphi$

$G, \sigma, n \models \exists \lambda \in \text{Paths}(G, n). \exists i. G, \sigma, \lambda_i \models \varphi_1 \land \forall j < i. G, \sigma, \lambda_j \models \varphi_1$

$G, \sigma, n \models \exists \lambda \in \text{RPaths}(G, n). \exists i. G, \sigma, \lambda_i \models \varphi_1 \land \forall j < i. G, \sigma, \lambda_j \models \varphi_1$

$G, \sigma, n \models \exists x. \varphi$ iff $\exists o. G, \sigma(x \mapsto o), n \models \varphi$

The set of atomic predicates used in conditions may depend on the target language under consideration, but some simple predicates are applicable to almost every language, and many optimizations can be specified with only language-independent predicates. These predicates include the following:

- $\text{is}(x, y)$ is true if $x$ and $y$ are the same object.
- $\text{node}(n)$ is true at a node $n'$ iff $n' = n$.
- $\text{stmt}(\ell)$ is true at $n$ if $L(n) = \ell$.
- $\text{out_edges}(s)$ is true at $n$ when $s = \{(d, m) \mid (n, d, m) \in E\}$ where $E$ is the set of edges of $G$.
- $\text{in}(n', d)$ is true at $n$ when $n$ has an incoming edge from $n'$ with label $d$ in $G$ (that is, $(n', d, n) \in E$).
• out\((d, n')\) is true at \(n\) when \(n\) has an outgoing edge to \(n'\) with label \(d\) in \(G\) (that is, \((n, d, n') \in E)\).

• start is true at \(n\) when \(n \in S\), that is, at any start node of \(G\).

• fresh_node\((n')\) is true when \(n'\) is a node not used in \(G\).

Note that all of these predicates are static properties of GCFGs that do not depend on the semantics of the language under consideration. Many Morpheus optimizations can be stated and performed independently of the semantics of the target language, so that Morpheus may serve as a design tool even in the absence of formal semantics for the target language.

### 3.4. Semantics of Transformations

The semantics of strategies is given by a function \([T]\)(\(\sigma, G\)) that takes a transformation, a substitution, and a GCFG and returns the set of GCFGs that can be produced by the transformation. In order to give semantics to the \(*\) strategy, we must define the result of applying a transformation to a graph some finite (but unbounded) number of times:

**Definition 6.** Let \(F\) be a function mapping a pair of a substitution and a GCFG to a set of GCFGs. Then, we define the function \texttt{apply\_some} by the following inductive rules:

\[
G \in \texttt{apply\_some}(F, \sigma, G) \quad G' \in F(\sigma, G) \quad G'' \in \texttt{apply\_some}(F, \sigma, G')
\]

\[
G'' \in \texttt{apply\_some}(F, \sigma, G)
\]

If we view \(F\) as a relation on GCFGs, then \texttt{apply\_some} gives its reflexive transitive closure.

With this, the semantics of strategies is defined as follows:

**Definition 7.** Let \(T\) be a transformation, \(\sigma\) a substitution mapping metavariabes to values, and \(G\) a GCFG. The semantics of a transformation is given
by the function $[T](\sigma, G)$ defined recursively as follows:

$$[A](\sigma, G) = \begin{cases} \{ A(A)(\sigma, G) \} & \text{if } A(A)(\sigma, G) \text{ is defined} \\ \{ \} & \text{otherwise} \end{cases}$$

$$[\text{SATISFIED_AT } n \varphi](\sigma, G) = \begin{cases} \{ G \} & \text{if } G, \sigma, \sigma(n) \models \varphi \\ \{ \} & \text{otherwise} \end{cases}$$

$$[\text{NOT } T](\sigma, G) = \begin{cases} \{ G \} & \text{if } [T](\sigma, G) = \{ \} \\ \{ \} & \text{otherwise} \end{cases}$$

$$[T_1 \setminus T_2](\sigma, G) = [T_1](\sigma, G) - [T_2](\sigma, G)$$

$$[\text{EXISTS } x. T](\sigma, G) = \{ G' \mid \exists o. G' \in [T](\sigma(x \mapsto o), G) \}$$

$$[T_1 + T_2](\sigma, G) = [T_1](\sigma, G) \cup [T_2](\sigma, G)$$

$$[T_1 ; T_2](\sigma, G) = \bigcup_{G' \in [T_1](\sigma, G)} [T_2](\sigma, G')$$

$$[T^*](\sigma, G) = \text{apply_some}([T], \sigma, G)$$

Note that $\text{EXISTS}$ may quantify over any program object, whether present in the graph or not, so that transformations can introduce fresh nodes and numerical values that do not appear in the original program. Note also that if $\sigma$ is an ill-typed substitution—e.g., it associates a program expression to a metavariable meant to represent a node—then $[T](\sigma, G)$ will be the empty set. Since Morpheus transformations will always be evaluated starting with an empty substitution, this is the correct behavior: an ill-typed substitution represents a failed search branch rather than user error.

The above language is quite expressive, and for the sake of conciseness it is convenient to introduce a collection of derived strategies:

$$\text{Id} = \text{EXISTS } n. \text{SATISFIED_AT } n \text{ true}$$

$$\text{Fail} = \text{EXISTS } n. \text{SATISFIED_AT } n \text{ false}$$

$$\text{ALLOW } T = \text{NOT } (\text{NOT } T)$$

$$\text{APPLY_ALL } T = T^* ; \text{NOT } (T \setminus \text{Id})$$
The semantics of these derived constructs then are:

\[
\begin{align*}
\langle \text{Id} \rangle(\sigma, G) &= \{G\} \\
\langle \text{Fail} \rangle(\sigma, G) &= \{\} \\
\langle \text{ALLOW} T \rangle(\sigma, G) &= \begin{cases} 
\{G\} & \text{if } \langle T \rangle(\sigma, G) \neq \{\} \\
\{\} & \text{otherwise}
\end{cases} \\
\langle \text{APPLY\_ALL} T \rangle(\sigma, G) &= \begin{cases} 
\text{apply\_some}(\langle T \rangle(\sigma, G)) - \{G' \mid \exists G'' \neq G'. G'' \in \langle T \rangle(\sigma, G)\} & \text{if } \langle T \rangle(\sigma, G) \neq \{\} \\
\{\} & \text{otherwise}
\end{cases}
\end{align*}
\]

In particular, \text{APPLY\_ALL} T returns all graphs produced by repeatedly applying T to the original graph until it is no longer applicable.

3.5. Strategy Semantics as a Kleene Algebra

The strategies of Morpheus presented above allow for transformations built from application of actions combined with sequencing, choice, repetition, and some other constructs. This structure is that of a Kleene algebra:

**Definition 8.** A Kleene algebra [4] is a base set \( K \) with two special elements 0 and 1, and three operations +, \( \cdot \) and \( * \), such that \((K, 0, 1, +, \cdot)\) is a semiring idempotent in +, and such that

- \( 1 + (t \cdot t^*) = 1 + (t^* \cdot t) = t^* \)
- \( s + (t \cdot r) + r = r \Rightarrow (t^* \cdot s) + r = r \)
- \( s + (r \cdot t) + r = r \Rightarrow (s \cdot t^*) + r = r. \)

The idempotent operator + gives rise to a partial order on \( K \) defined as \( s \leq t \equiv s + t = t. \) The last equations may then be restated as \( s + (t \cdot r) \leq r \Rightarrow (t^* \cdot s) \leq r \) and \( s + (r \cdot t) \leq r \Rightarrow (s \cdot t^*) \leq r. \)

Kleene algebras generalize regular expressions and have an equational theory similar to what one would expect based on one’s intuition about regular expressions; for example, \((T^*)^* = T^*\). This not only makes it easy for programmers to use familiar intuitions to reason about these transformations, but also equips the language with a rich theory to assist in proving properties of transformations.

At the purely syntactic level, the strategy language of Morpheus cannot form a Kleene algebra because syntactic equality is purely structural equality. A Kleene algebra for this language exists only at the semantic level.

The following lemma expands on the semantics of \( T^* \):
Lemma 1. Let $T$ be a transformation, $\sigma$ be a total substitution, and $G, G'$ be GCFGs. Then the following two assertions are equivalent:

1. $G' \in [T*](\sigma, G)$
2. $G' = G$ or there exists $n > 0$ such that $G' \in \left[ T ; \ldots ; T \right]_n(\sigma, G)$

Proof. That 2 implies 1 follows directly from induction on $n$ and the definition of apply_some. That 1 implies 2 follows by rule induction on the definition of apply_some. $\square$

Now we can state and prove the sense in which the strategy language of Morpheus gives us a Kleene algebra.

Theorem 1. Let $K = \{ F \mid \exists T'. T' a strategy in Morpheus and F = [T']\}$. Then $(K, [\text{Fail}], [\text{Id}], \lambda T_1 T_2. [T_1 + T_2], \lambda T_1 T_2. [T_1 ; T_2], \lambda T. [T*])$ is a Kleene algebra.

Proof. That $\lambda T_1 T_2. [T_1 + T_2]$ is associative, commutative, and idempotent with $[\text{Fail}]$ as its identity follows from the fact that union is associative, commutative, and idempotent with $\{ \}$ as its identity. The operation $\lambda T_1 T_2. [T_1 ; T_2]$ is in essence relation composition restricted to the semantics of transformations, which is associative with the identity function as its left and right identity, and the empty relation its left and right annihilator. Similarly, the distributive properties follow from the facts that mapping a relation over the union of two sets is equivalent to taking the union of the result of mapping the relation over each, and taking the union of mapping two relations over a set is equivalent to mapping the union of two relations over the set. Therefore $(K, [\text{Fail}], [\text{Id}], \lambda T_1 T_2. [T_1 + T_2], \lambda T_1 T_2. [T_1 ; T_2])$ forms a semiring. That $[\text{Id} + T ; T*] = [T*]$ follows immediately from the definition of apply_some. On the other side, $[\text{Id} + T* ; T] = [T*]$ can be seen with a pointwise argument using Lemma 1.

For the last two properties, note that the ordering generated by $\lambda T_1 T_2. [T_1 + T_2]$ gives us that $[T_1] \leq [T_2]$ iff for every GCFG $G$ and total substitution $\sigma$ we have $[T_1](\sigma, G) \subseteq [T_2](\sigma, G)$, the usual ordering on functions into sets. Also, notice that $\lambda T_1 T_2. [T_1 + T_2]$ and $\lambda T_1 T_2. [T_1 ; T_2]$ monotonic in each argument, and that $[T_1] \leq [T_1 + T_2]$ and $[T_2] \leq [T_1 + T_2]$.

Thus, to finish the proof, it suffices to show for all transformations $S, T$, and $R$ that
For each property, assuming the antecedent of the implication, we have \( [S] \leq [R] \). With property 1, we also have the \( [T; R] \leq [R] \) and hence \( [T; S] \leq [S] \), and by induction \( [T; \ldots; T; S] \leq [S] \) for all \( n > 0 \).

To show that \( [T*; S] \leq [R] \) it suffices to show that for all \( \sigma, G, \) and \( G' \), if \( G' \in [T*; S](\sigma,G) \), then \( G' \in [R](\sigma,G) \). Fix \( G' \in [T*; S](\sigma,G) \). Now, \( [T*; S](\sigma,G) = \{ G'' | \exists G'' \in [T*](\sigma,G) \}. G'' \in [S] \} \). Therefore, there exists \( G'' \in [T*](\sigma,G) \) such that \( G' \in [S](\sigma,G'') \). Fix such a \( G'' \). Because \( G'' \in [T*](\sigma,G) \), by Lemma 1, either \( G'' = G \) or there exists an \( n > 0 \) such that \( G'' \in [T; \ldots; T](\sigma,G) \). If \( G'' = G \), then \( G' \in [S](\sigma,G) \) and since \( [S] \leq [R] \), we have \( G' \in [R](\sigma,G) \) as was to be shown in property 1. So assume \( G'' \in [T; \ldots; T](\sigma,G) \) for some \( n > 0 \).

Then, since \( [T; \ldots; T; S] \leq [S] \) again we have \( G'' \in [S](\sigma,G) \), and hence \( G' \in [R] \) finishing property 1. The proof for property 2 is similar.

4. Executable Semantics for Conditions and Strategies

The semantics of conditions and strategies given in the previous section is in terms of a generalized control flow graph and a total substitution. In particular, a model for an FOCTL formula comprises a GCFG, a node in the GCFG, and a substitution. When applying a Morpheus transformation to a GCFG, the problem arises of finding substitutions that satisfy the FOCTL formulae in the transformations at the specified nodes. Following the mathematical semantics of Morpheus, we show how the problem of finding satisfying substitutions for an FOCTL formula relative to a given GCFG and node can be reduced to finding satisfying substitutions for a derived first-order formula. We conclude the section by showing how to use this reduction, together with SMT solving, to give an executable semantics for Morpheus.
4.1. FOCTL Model Finding

The model checking problem for CTL and its variants is a well-studied problem with a well-known efficient algorithm [5], but considerably less attention has been given to the analogous FOCTL problem of model finding. The model finding problem in its general form, as described by Bohn et al. [6], is this: given an FOCTL formula $\varphi$ built from a set of atomic predicates where the predicates may contain free variables, a transition system $S$, and an interpretation of the atomic predicates on $S$, what are the possible assignments of values to the free variables of $\varphi$ such that $\varphi$ holds on $S$? When a formula contains no free variables, model finding is simply model checking; in the general case, it is considerably more complex.

Following Bohn et al., we present an algorithm for FOCTL model finding by reducing the combination of FOCTL formulae and GCFGs to first order formulae and turning these over to an SMT solver to generate the possible solving substitutions. We differ from Bohn et al. in our treatment of formulae of the type $E \varphi_1 U \varphi_2$ and $A \varphi_1 U \varphi_2$, in handling past-time operators, and in having a full implementation. Our algorithm is given in a functional style and can be straightforwardly implemented in a functional programming language. We find satisfying models symbolically, by defining a function Satis that, given a formula $\varphi$ and a node $n$, constructs a non-temporal first-order formula characterizing the set of substitutions that make $\varphi$ true at $n$.

The following theorem states the correctness of the algorithm:

**Theorem 2.** Let $G = (N, L, S, E)$ be a GCFG, $n \in N$ and $\varphi$ a FOCTL formula. Then \( \{ \sigma \mid G, \sigma, n \models \varphi \} = \{ \sigma \mid \sigma \models_{FOL} \text{Satis}(\varphi)(n) \} \).

This theorem is proved by induction on the lexicographic order of the number of $A \varphi_1 U \varphi_2$-headed subformulae of $\varphi$ and the number of subformulae of $\varphi$; we will define Satis and give the proof of its correctness case by case. To handle atomic predicates we assume that the language definition provides an interpretation $\llbracket p(\vec{x}) \rrbracket_n$ of each atomic predicate $p$ with arguments $\vec{x}$ at a node $n$. When the head connective is a non-temporal connective, Satis recursively translates its subformulae, leaving the connective untouched:

\[
\begin{align*}
\text{Satis}(p(\vec{x}))(n) &= \llbracket p(\vec{x}) \rrbracket_n \\
\text{Satis}(\varphi_1 \land \varphi_2)(n) &= \text{Satis}(\varphi_1)(n) \land \text{Satis}(\varphi_2)(n) \\
\text{Satis}(\neg \varphi)(n) &= \neg \text{Satis}(\varphi)(n) \\
\text{Satis}(\exists x. \varphi)(n) &= \exists x. \text{Satis}(\varphi)(n)
\end{align*}
\]

16
Correctness for these cases follows directly from the inductive hypothesis.

When $\varphi = \mathcal{E} \varphi_1 \mathcal{U} \varphi_2$, we need to ensure that we find a suitable witness path for the until-formula for each substitution. To do so, we define an auxiliary function $\text{Paths}_\varphi(I, F, k, n)$ that takes an invariant $I: N \to \text{FOL}$, a final requirement $F: N \to \text{FOL}$, an upper bound $k$, and a node $n \in N$, as follows:

$$
\begin{align*}
\text{Paths}_\varphi(I, F, 0, n) &= F(n) \\
\text{Paths}_\varphi(I, F, k, n) &= \text{Paths}_\varphi(I, F, k - 1, n) \\
&\quad \lor \left( I(n) \land \bigvee_{n' \in \text{succ}(E, n)} \text{Paths}_\varphi(I, F, k - 1, n') \right)
\end{align*}
$$

where $\text{succ}(E, n)$ is the stuttering invariant set of successors of $n$ in $\mathcal{E}$, i.e., $\{ n' \mid n = n' \lor (n, n', d) \in E \}$.

**Lemma 2.** $\text{Paths}_\varphi(I, F, k, n)$ characterizes the set of substitutions $\sigma$ such that there is a path $\lambda$ from $n$ and there exists $j \leq k$ along which $G, \sigma, \lambda_j \models F(\lambda_j)$ and $G, \sigma, \lambda_i \models I(\lambda_i)$ for all $i < j$.

We can then define

$$
\text{Satis}(\mathcal{E} \varphi_1 \mathcal{U} \varphi_2)(n) = \text{Paths}_\varphi(\text{Satis}(\varphi_1), \text{Satis}(\varphi_2), |N|, n)
$$

and finish the proof of this case by noting that, since if there is any witness there must be a cycle-free witness, Lemma 2 ensures the presence of a suitable witness.

When $\varphi = \mathcal{A} \varphi_1 \mathcal{U} \varphi_2$, we need to give a first order formula that expresses the property that every path forward eventually satisfies $\varphi_2$ and satisfies $\varphi_1$ at every point before that. To create such a formula, we search for witnesses to the until-formula, this time in a conjunctive fashion. To do this we define the auxiliary function $\text{Paths}_\lambda(I, F, k, n)$ that takes an invariant $I: N \to \text{FOL}$, a final requirement $F: N \to \text{FOL}$, a length $k$, and a node $n \in N$:

$$
\begin{align*}
\text{Paths}_\lambda(I, F, 0, n) &= F(n) \\
\text{Paths}_\lambda(I, F, k, n) &= \text{Paths}_\lambda(I, F, k - 1, n) \\
&\quad \lor \left( I(n) \land \bigwedge_{n' \in \text{succ}(E, n)} \text{Paths}_\lambda(I, F, k - 1, n') \right)
\end{align*}
$$
Lemma 3. $\text{Paths}_\lambda(I, F, k, n)$ characterizes the set of substitutions $\sigma$ such that for every path $\lambda$ from $n$ there exists $j \leq k$, s.t. where $G, \sigma, \lambda_i \models F(\lambda_i)$ and $G, \sigma, \lambda_j \models I(\lambda_j)$ for all $j < i$.

Unfortunately, while this gives us an ability to say that we can “always” find a witness for our until-formula $\varphi_2$ nearby, this function by itself still allows for infinite paths that never reach their satisfying witness. Specifically, the presence of cycles means that there could exist an infinite path that always satisfies $\varphi_1$ but never leaves its cycle, despite the fact that it needs to do so to find its $\varphi_2$ witness. Earlier, formulae of the form $\mathcal{E} \varphi_1 \mathcal{U} \varphi_2$ did not have to be concerned about avoiding loops because, if we found a witness, whether there was a loop or not, it was a witness and that was all we needed. Here, we need one more auxiliary function to help us avoid infinite counterexamples. This is defined as follows.

$$\text{Lasso}(I, 0, n) = \text{TRUE}$$
$$\text{Lasso}(I, k, n) = \bigvee_{n' \in \text{succ}(E, n)} (I(n') \land \text{Lasso}(I, k - 1, n'))$$

Lemma 4. $\text{Lasso}(I, k, n)$ characterizes the set of substitutions $\sigma$ such that there is a path $\lambda$ from $n$ along which $G, \sigma, \lambda_i \models I(\lambda_i)$ for every $i < k$.

We can then define

$$\text{Satis}(\mathcal{A} \varphi_1 \mathcal{U} \varphi_2)(v) = \neg \text{Lasso}(\text{Satis}(\varphi_1 \land \neg \varphi_2), |N| + 1, n) \land \text{Paths}_\lambda(\text{Satis}(\varphi_1), \text{Satis}(\varphi_2), |N|, n)$$

We use our two auxiliary functions to ensure that every path from $n$ has a suitable witness for $\varphi_2$ and that, by the pigeonhole principle, it has no paths along which $\varphi_1$ holds and $\varphi_2$ is never reached (i.e., it has no cycles satisfying $\varphi_1$ but not $\varphi_2$). This case demonstrates why simple induction on the size of $\varphi$ is not sufficient to prove correctness; $\text{Satis}(\mathcal{A} \varphi_1 \mathcal{U} \varphi_2)(n)$ makes recursive calls on formulae that are not strictly smaller than $\mathcal{A} \varphi_1 \mathcal{U} \varphi_2$, but those formulae have fewer $\mathcal{A} \mathcal{U}$ connectives.

In handling the past-time connectives we have a slight advantage: since all paths backward must eventually reach a start node of the graph, we can ignore the possibility of paths with infinite cycles. We make an additional simplifying assumption that all nodes in the graph are reachable from the start node (unreachable nodes can be safely discarded). This allows
us to handle the backwards cases with the duals of $\text{Paths}_\lor$ and $\text{Paths}_\land$ formed by following predecessors instead of successors, denoted $\text{Paths}_\lor$ and $\text{Paths}_\land$ respectively, where the set of predecessors of a node $n$ is defined as $\text{pred}(E, n) = \{n' \mid n = n' \lor (n', n, d) \in E\}$.

$$\text{Satis}(E \varphi_1 S \varphi_2)(n) = \text{Paths}_\lor(\text{Satis}(\varphi_2), \text{Satis}(\varphi_1), |N|, n)$$

$$\text{Satis}(A \varphi_1 S \varphi_2)(n) = \text{Paths}_\land(\text{Satis}(\varphi_2), \text{Satis}(\varphi_1), |N|, n)$$

This completes the definition of the Satis function. The running time of the algorithm as written is $O\left(|N|^{|N||\varphi|}\right)$, but we can do better using dynamic programming. We begin with a table of $O(|\varphi||N|)$ entries for Satis and tables of size $O(|N|)$ for the each of the path-searching functions that represent the results for each possible input given the size of $G$ and the subformulae of $\varphi$. Filling each of the path tables (assuming their arguments are already evaluated) takes $O(|N|^3)$ time, and the tables can be reused to answer their queries for all nodes. Thus, the amortized running time for filling an entry in the table for Satis is $O(|N|)$, and the overall reduction runs in $O(|\varphi||N|^2)$ time. Once we have the characteristic formula provided by Satis, if the basic language of atomic predicates is amenable to SMT solving, we can use a solver to compute the concrete set of satisfying models.

4.2. Executable Semantics for Strategies

Given the model finding algorithm described above, we can define a function $\text{models}_\text{of}(G, n, \varphi)$ that computes a stream of partial substitutions representing the satisfying models of $\varphi$. We do so by first using Satis to generate a first-order formula with the same satisfying substitutions as $\varphi$, and then using an SMT solver to find all satisfying models of that formula. We will represent streams here as lists, with the understanding that they may be finite or infinite. Theorem 2 then assures us that

$$\bigcup_{\tau \in \text{models}_\text{of}(G, n, \varphi)} \{\sigma \mid \sigma|_{\text{dom}(\tau)} = \tau\} = \{\sigma \mid G, n, \sigma \models \varphi\},$$

and so $\text{models}_\text{of}$ serves as an executable method for finding satisfying models of Morpheus conditions. To find only models that match some already-known partial substitution, we also define a function $\text{get_models}(G, n, \varphi, \tau)$ that returns only the results of $\text{models}_\text{of}(G, n, \varphi)$ that are extensions of $\tau$. Using $\text{get_models}$, we can write an executable function trans_sf that takes a transformation and a pair of a partial substitution of the values for metavariables.
learned so far and a GCFG, and returns a stream of pairs of an extended partial substitution and a transformed graph. Recall that the abstract semantic function for an action on a GCFG and a total substitution, $\langle A \rangle(\sigma, G)$, is already executable. We will extend this to use a partial substitution $\tau$ by having $\langle A \rangle(\tau, G)$ raise an exception (as distinct from returning a value symbolizing an undefined result) when $\tau$ fails to provide a value for any of the free metavariables needed by $A$. For a GCFG $G$, let $\text{Nodes}(G)$ be a list of the nodes in $G$. With these we can define $\text{trans}_\text{sf}(T) (\tau, G)$ (in ML-like pseudocode) as follows:

$$\text{trans}_\text{sf}(A) (\tau, G) = \begin{cases} [\tau, \langle A \rangle(\tau, G)] & \text{if } \langle A \rangle(\tau, G) \text{ is defined} \\ [] & \text{otherwise} \end{cases}$$

$$\text{trans}_\text{sf}(\text{SATISFIED}_\text{AT} n \varphi) (\tau, G) = \begin{cases} \text{map}(\lambda t. (t, G))(\text{get}_\text{models}(G, n, \varphi, \tau)) & \text{if } \tau(n) \in \text{Nodes}(G) \\ \text{flatten} & \end{cases}$$

$$\text{trans}_\text{sf}(\lambda m.(\text{map}(\lambda t. (t, G))(\text{get}_\text{models}(G, m, \varphi, \tau + \{n \mapsto m\}))) \text{ (Nodes}(G)))$$

$$\text{trans}_\text{sf}(\text{NOT} T) (\tau, G) = \text{if } \text{trans}_\text{sf}(T) (\tau, G) = [] \text{ then } [\tau, G] \text{ else } []$$

$$\text{trans}_\text{sf}(\text{EXISTS} x. T) (\tau, G) = \begin{cases} \text{let } y = \text{fresh}_\text{for}(x, \text{dom}(\tau) \otimes \text{free}_\text{vars}(G)) \\ \text{in } \text{trans}_\text{sf}(\text{subst}(x, y) T) (\tau, G) \end{cases}$$

$$\text{trans}_\text{sf}(T_1 \setminus T_2) (\tau, G) = \text{filter}(\lambda r. r \notin \text{trans}_\text{sf}(T_2) (\tau, G))(\text{trans}_\text{sf}(T_1) (\tau, G))$$

$$\text{trans}_\text{sf}(T_1 ; T_2) (\tau, G) = \text{flatten}(\text{map}(\lambda r. \text{trans}_\text{sf}(T_2) r)(\text{trans}_\text{sf}(T_1) (\tau, G)))$$

$$\text{trans}_\text{sf}(T_1 + T_2) (\tau, G) = \text{trans}_\text{sf}(T_1) (\tau, G) \circ \text{trans}_\text{sf}(T_2) (\tau, G)$$

$$\text{trans}_\text{sf}(T *) (\tau, G) = [(\tau, G)] \circ (\text{flatten}(\text{map}(\lambda r. \text{trans}_\text{sf}(T *) r)(\text{trans}_\text{sf}(T) (\tau, G))))$$

It is worth noting that the executable semantics for $\text{trans}_\text{sf}(T *) (\tau, G)$ is a direct implementation of the Kleene Algebra law: $\text{trans}_\text{sf}(T *) = \text{trans}_\text{sf}(\text{Id} + (T ; (T *) )).$
The definition of \texttt{trans} as an executable function can only approximate a faithful representation of infinite results. One compromise was already mentioned: for actions, we require that the partial substitution supply all needed values rather than creating all possible total substitutions completing the given partial substitution. In the description above, each operation combining the results of two transformations requires the component results to be finite. If we use streams instead of true finite lists, then we can preserve more of the behavior in the case of infinite results. The functions \texttt{map}, \texttt{flatten} and \texttt{@} may all be done lazily, so that results may be produced one at a time. This still leaves us with a problem in the case of $T_1 \setminus T_2$. Here, the removal of the results of $T_1$ that could have produced by $T_2$ can be done one at a time for $T_1$, but must have all the results of $T_2$ available, requiring there to be only finitely many such results. However, as long as \texttt{trans} terminates, it can be shown that \texttt{trans}(T) $(\tau, G) = [T](\tau, G)$, and so \texttt{trans} is a (partially) correct executable semantics for Morpheus transformations.

This gives us an algorithm for computing the result graphs for a given transformation, which can be implemented in a functional language (we have chosen F# for its Z3 integration). As long as a transformation expressed in Morpheus does not require infinite computations, we can run it on a target graph and obtain all of its outputs. In Section 6, we demonstrate the use of these semantics to define, test, and refine a sample optimization.

5. Implementation

We have implemented the executable semantics of Morpheus described above in F# [7], taking advantage of its integration with the Z3 SMT solver [8]. The semantic functions for actions and strategies in Section 4.2 can be straightforwardly translated into F# code. We use the algorithm of Section 4.1 to reduce side conditions to first-order formulae, for which Z3 can generate satisfying models. In order to find every satisfying model and generate all possible results of a transformation, we make repeated calls to Z3, adding conditions that precisely rule out the models that have already been found. We memoize the \texttt{Satis} function with a standard lookup table in order to achieve the improved running time. The examples of Section 6.4 complete in between 1 and 4 seconds, with the majority of the running time devoted to constructing SMT queries; we believe that further optimization of the condition-generation process will allow the semantics to scale to more extensive program graphs.
One of the most important features of Morpheus as a program transformation language is its *language-independence*. Because the definition of Morpheus does not depend on the language of instructions in the graph being transformed, we can use it to transform graphs labeled with instructions in any language with the correct structure. This is not only a theoretical property of the language definition but also a feature of our implementation: we can use F#'s reflection feature to automatically execute Morpheus formulae on graphs in any language, simply by giving a datatype for the syntax of the language.

To use a language as a target for Morpheus, we must have access to four elements:

1. The syntax of concrete instructions (as shown for MiniLLVM in Section 6.2).

2. The syntax of instruction *patterns*, in which some elements may be replaced by metavariables. These patterns are used in the body of Morpheus transformations on the language, as in the RSE transformation in Section 6.4. Thus far we have assumed that the idea of “replacing parts of instructions with metavariables” is straightforward, but it is not a built-in feature of F# (or of most other programming languages).

3. A substitution function that takes an assignment of program objects to metavariables and applies it to a pattern to yield a concrete instruction.

4. A two-way translation between instructions, patterns, and their components on the one hand and terms in Z3 on the other, so that we can use Z3 to check atomic predicates involving instructions (such as `stmt(p)`)) and apply the results to program graphs.

At least one of 1 and 2 is absolutely necessary, since we must know the syntax of the language. However, once we have the syntax, the remaining elements follow fairly mechanically. This should allow us to define them once for all languages, so that all we need to do to add a new target language is to define its syntax. One approach would be to define terms, patterns, and substitution at an abstract level (as F# datatypes and functions), and give the syntax of a target language as an object of a term datatype. However, this would introduce an extra level of complexity when defining languages, forcing users who want to add a language to understand not just F# syntax.
but also our custom datatype. This approach would also ignore an advantage that we already have available: patterns and substitution are fundamental objects in Z3. A term such as `alloca t` where `t` is a variable may not be a valid MiniLLVM instruction, but it is a valid Z3 expression, and Z3 can substitute a concrete MiniLLVM type for `t`. Thus, if we can convert the syntax of instructions and patterns into Z3 sorts and back, we can perform substitution simply by translating a pattern into Z3, calling Z3's substitution, and then translating the result back into a concrete instruction.

Fortunately, both F# and Z3 speak the language of recursive datatypes. Given a type defined by a set of constructors – such as a BNF grammar for the syntax of a language – in F#, we can recursively construct a corresponding Z3 datatype sort with the same constructors. This gives us a straightforward way to translate between F# and Z3. Given an F# object that is a constructor applied to a list of arguments, we use reflection to find its type and type constructors, construct the corresponding sort in Z3, translate the F# constructor to the corresponding Z3 constructor, and then recursively translate the arguments. In practice, we use a type table that maps each F# type to its corresponding Z3 sort, and reuse existing Z3 sorts whenever possible, so that two instructions translate into the same Z3 instr sort; then to translate back from Z3 to F#, we need only do a reverse lookup in the table and translate the constructors in reverse.

Crucially, by structuring our syntax definition to indicate the places where metavariables can occur and ignoring these markers when translating into Z3, we can translate patterns and concrete instructions to the same sort in Z3, allowing us to write and solve equations such as `alloca t = alloca int`. While from F#'s perspective these instructions have two different types, Z3 understands that `t` is being used as a metavariable in a term that represents a concrete instruction. Then in order to use the executable semantics of Morpheus with a new target language, a user needs only to define the syntax of concrete instructions as an F# datatype, with a few markers indicating the places where metavariables may occur. Our executable semantics automatically derives a translation into and out of Z3 and the resulting notion of substitution. We have tested this in practice by adding a subset of Microsoft’s Common Intermediate Language [9, 10] as a second target; after giving the grammar of the language in terms of F# types, we were immediately able to begin writing and executing Morpheus transformations on it, just as we did for MiniLLVM, with no change to the existing code.

In addition to the target language, Morpheus is parameterized by the set
of atomic predicates that need to be translated into first-order predicates in order for Satis to be able to generate satisfying substitutions for the conditions in the Morpheus transformations. Our implementation includes a small choice of such functions, including one translating the atomic predicates used in the examples in this paper. When these are combined with patterns for graph components like instructions and edge labels, together with the first order logic inherent in FOCTL, we can express all the predicates we have needed to date. Should there be new properties of the graph that do not fall within this set, but can be expressed in Z3, it is possible for the advanced user to write their own F# function translating new atomic predicates into Z3, and supply it as an argument to the function get_models and through that to Satis.

6. Designing and Prototyping Optimizations with Morpheus

Morpheus is a parameterized language for the specification, testing, and verification of the safety of program transformations performed on programs represented as control flow graphs. In this section we will explore some of the range of application of Morpheus with examples. First we will examine some of the versatility of the language with examples of transformations that are independent of the language of the programs being transformed. Next we examine transformations that are to some degree language dependent and give examples of several transformations expressed at the language-dependent level. Finally, we will discuss the use of the executable semantics of Morpheus in incrementally developing a transformation, showing how testing can help with developing the necessary conditions permitting the transformation to be applied safely.

6.1. Some Simple Examples of Morpheus Transformations

It is possible to write in Morpheus some simple transformations that are safe (i.e., generate programs that are functionally equivalent to the input program) independent of the target language. Possibly the simplest of these is the removal of a node that is not a start node and that has no edges coming in to it. The semantics of remove_node guarantees that removing any node that is not a start node of the graph will not alter the executions of the represented program, assuming that all threads start their execution at start nodes. We could leave the requirement that the node have no incoming edges implicit, to be checked by the semantics of remove_node, but it is better
documented code to check it explicitly. Thus, we can define a safe node removal as follows:

\[
safe\_remove\_node(n) \triangleq SATISFIED\_AT\ n (\neg start \land \neg (\exists m. \exists d. in(m, d))) ; remove\_node(n)
\]

By parameterizing this by the node to be deleted, we may use this transformation in conjunction with others in specific locations. However, we may also use it to define the removal of all non-start nodes with no incoming edges as follows:

\[
delete\_inaccessible\_nodes \triangleq (EXISTS n. safe\_remove\_node(n))^* ; NOT EXISTS n. SATISFIED\_AT\ n (\neg start \land \neg (\exists m. \exists d. in(m, d)))
\]

The transformation delete_inaccessible_nodes displays a specification paradigm that will often be repeated in examples below. At the core we specify a single step of a transformation. This base transformation is preceded by a condition. The variables shared between the condition and the transformation allow the condition to pass specific values to the transformation. This sequence is then prefixed by an existential, allowing the conditional transformation to be applied in different circumstances where the condition is satisfied. Finally, the resultant transformation is performed repeatedly until a termination condition is satisfied, such as the enabling condition no longer being satisfiable. This paradigm is similar to that of a while loop in imperative programming, but can also be used in other settings where the enabling condition principally serves to supply values to the inner transformation and has no direct link to the termination condition.

Another safe transformation we may perform independently of the underlying programming language is to move an edge’s target from one node to another provided the instruction on each of the target nodes is the same and the outgoing edges point to the same nodes via the same labels:

\[
safe\_move\_edge((n, d, m_1), m_2) \triangleq \\
(EXISTS l s. ((SATISFIED\_AT m_1 (stmt(l) \land out\_edges(s))) ; (SATISFIED\_AT m_2 (stmt(l) \land out\_edges(s)))) ; move\_edge((n, d, m_1), m_2)
\]

This may not seem like much of an optimization in itself, but we will see its use in loop peeling below.
Another example of a simple language-independent transformation is node copying:

\[
\text{copy_node}(n, n') \triangleq \\
\exists l s. (\text{Satisfied}_n(l) \land \text{out_edges}(s) \land \text{fresh_node}(n')) \\
\text{add_node}(n', l, s)
\]

Using \textit{safe_move_edge} and \textit{copy_node}, we can build a transformation for copying loops that do not have merge points. To do so, we must first have conditions that can recognize properties like “there is a loop from one point that passes through another”, and “all nodes internal to all loops from a point to itself have only one incoming edge”.

\[
\text{has_loop_through}(m) \triangleq \\
\exists n. \text{node}(n) \land \text{EX} (\text{EF} (\text{node}(m) \land \text{EF node}(n)))
\]

\[
\text{entry_for_loop_through}(m) \triangleq \\
\exists n. \text{node}(n) \land \text{has_loop_through}(m) \land \text{EX} \neg \text{has_loop_through}(n)
\]

\[
\text{unique_in_edge_on_loops} \triangleq \\
\exists n. \text{node}(n) \land \\
\neg (\exists m_1 m_2 d_1 d_2. \text{EX} (\neg \text{node}(n) \land \text{in}(m_1, d_1) \land \text{in}(m_2, d_2) \land \\
\neg \text{is}(m_1, m_2) \land \text{EF node}(n)))
\]

The test \text{has_loop_through}(m) asks whether, after taking one step forward, we can reach \( m \), and then carry on to reach again the node at which we are doing the test. An interesting subcase is when we are already at the node \( m \). In this case, we are asking if there is a loop of length at least one from the current node back to itself.

For a node \( n \) to be an entry to a loop through \( m \), it must have a loop (of length at most one) through \( m \) back to \( n \). But we also need to have at least one edge that is truly an entry point to the loop. Along the loop through \( m \) there may be branches, and some of those branches may well lead back to \( n \), but we do not wish to view these edges as true entry points. Thus, we also require there to be a predecessor to \( n \) that is not involved in any loop back to \( n \), which gives us exactly the definition of \text{entry_for_loop_through}(m).

For purposes of program transformations, when we refer to copying a loop, we are referring to a loop in the programming sense rather than the strict
graph theoretic sense, in that we wish to allow branching structure within the loop, and not restrict ourselves to copying a single path, but rather to copying every path from the loop entry back to itself. In the most general case, this is handled by keeping a record of the nodes visited and checking the record before deciding whether to copy a given node. However, if every edge other than the chosen entry point has only one incoming edge, then we can copy the loop by walking forward through the loop, touching the successors of each node copied only once. The test unique_in_edge_on_loops when performed at the chosen loop entry performs the necessary checks on incoming edges. A simple example of programs satisfying the criterion are loops with unique entry points and straight-line (non-branching) bodies. More generally, we can allow branching so long as all branches only either exit the loop or merge at the entry point.

With these predicates in hand, we may now describe the procedure for copying the kind of loops described above. If \( n \) is the entry to a loop and every inner node on every loop from \( n \) to itself has only one predecessor, then make a copy of \( n \), move every edge that points into \( n \) from a node that does not have a path back to \( n \), then repeatedly copy every node that is an entry to a loop that passes through \( n \) that is not \( n \) (these nodes can only be nodes pointed to by freshly created nodes) and move all edges that point to the node copied that do not have a path back to \( n \) over to the copy.
The above transformation has moderately complex conditions describing which nodes to copy and which edges to move. These conditions are necessary to guarantee termination of the procedure and to achieve the desired result. It is worth noting, however, that in this case the safety of the program is already guaranteed by the general safety of `copy_node` and `safe_move_edge`. In subsequent transformations, we will pursue a similar approach of decomposition of transformations into fragments which are individually seen to be safe. This approach has its limits, particularly as the transformations become more language dependent, but it remains a useful objective.

6.2. A Sample Target Language: MiniLLVM

In the previous section (6.1), we gave examples of transformations that could be specified independently of the language of the programs being transformed. The transformations we will consider hereafter are dependent, at least to some degree, upon the language of the program being transformed. To support these transformations, we begin by defining a target language: MiniLLVM, a simplification of the LLVM intermediate language [11]. The instruction labels of MiniLLVM are defined as follows:

\[
\text{expr ::= } \%x \mid @x \mid c \quad \text{type ::= } \text{int} \mid \text{type}^*
\]
MiniLLVM expressions are either local variables (%x), global variables (@x), or constants. Instructions include arithmetic operations (where op is an arithmetic operator), comparison operations (where cmp is a comparison operator), conditional and unconditional branches, function calls and returns, memory allocation, loads from and stores to memory, and is pointer, which checks whether a given expression is pointer-valued (for use in loads and stores). (Note that the *’s indicate not repetition but pointer types.) Because the targets of control-flow instructions are encoded in the edges of the GCFG, the label arguments to br instructions and function names in call instructions are omitted.

We assume that multiple threads may simultaneously execute different portions of a MiniLLVM program, communicating via shared memory—although each alloca instruction is executed by a single thread, the memory allocated can be exposed to other threads by storing its location in a global variable or fixed memory location. In the following examples, we reason about concurrent execution with a sequential consistency memory model: i.e., in each step one thread in the program executes, and any memory operations immediately update the shared memory and are visible to all other threads. More relaxed memory models, such as total or partial store ordering, are important to consider when designing compiler optimizations on parallel programs, and any operational memory model can be integrated into the semantics of MiniLLVM (and thus the optimization testing process) with little difficulty.

6.3. Writing Morpheus Optimizations over MiniLLVM

Now that we have a target language, we can begin to define Morpheus transformations on MiniLLVM. The first set of examples use the br instruction. For MiniLLVM, this is the same as a SKIP instruction. It alters the program counter, but otherwise always succeeds and alters no local or global variables. In the context of a control flow graph, such an instruction may be eliminated by redirecting the edges coming in to it to its successor, and then deleting it.
skip_deletion(n) ≜
  (∃ m. (SATISFIED_AT n (stmt(br) ∧ out(seq, m))) ;
  (∃ n’ d. (SATISFIED_AT n in(n', d)) ;
     (move_edge((n’, d, n), m)) * ) ;
  SATISFIED_AT n ¬∃ n’ d. in(n', d)) ;
safe_remove_node(n)

The previously defined safe_move_edge cannot be used in this transformation, because the instruction to which we are moving the edges may not be labeled with br. The justification for this transformation is based on the dynamic semantics of the br instruction.

While the inverse, namely skip insertion, may seem odd as an “optimization”, it is also a safe transformation. Here, there is a question of where we wish to perform the skip insertion. Since the skip instruction must point somewhere, we can use the node that is to be its successor to describe where to put the skip. However, there is still a question of which edges are to be moved to point to the skip. We may move any subset of the nodes that point to the node that is to become the successor of the skip. There are at least three useful possibilities: we move a specified edge, effectively inserting the skip in the middle of it; we move all nodes pointing to the successor; or we move a specified subset of the edges. This set of choices highlights that there is a simpler transformation lurking inside skip insertion, that of moving an edge from after a skip to before it.

move_edge_to_skip((n, d, m), n') ≜
  SATISFIED_AT n' (stmt(br) ∧ out(seq, m)) ;
  SATISFIED_AT n out(d, m) ; move_edge((n, d, m), n')

Using this transformation we can then express skip insertion by specifying the strategy for selecting the edges to be moved to the inserted skip node. For example, we can specify the version of skip insertion that places it in the middle of a single edge as follows:

skip_insert_in_edge(n, d, m, n') ≜
  SATISFIED_AT n (out(d, m) ∧ fresh_node(n')) ;
  add_node(n', br, [(seq, m)]) ;
  move_edge_to_skip((n, d, m), n')
By using a strategy that moves all edges pointing to the successor of the skip being inserted, we can make a transformation for inserting a skip immediately before a node as follows:

\[
\text{skipinsert_before_node}(n, n') \triangleq \\
\text{SATISFIED AT } n (\text{fresh_node}(n')) ; \\
\text{add_node}(n', \text{br}, [(\text{seq}, n)]) ; \\
(\exists m d. \text{SATISFIED AT } n (\text{in}(m, d))) ; \\
\text{move_edge_to_skip}((n, d, m), n' \star)) ; \\
\text{SATISFIED AT } n (\neg \exists d. \exists m. \in (m, d))
\]

In each of these cases, since both of \text{add_node} and \text{move_edge_to_skip} transform programs to ones that are equivalent, both of the above transformations built from them also transform programs to equivalent ones. It is worth noting that a similar decomposition is possible for \text{skip_deletion}.

The transformations just introduced rely upon the underlying programming language having an instruction with “skip” semantics, altering only the program counter. For the next set, we will assume we have a syntactic test \text{def} to determine whether a given node (through its instruction) might alter the value of a variable, and a syntactic test \text{uses} to determine whether it might depend upon the variable. We can implement these predicates in MiniLLVM by defining \text{def}(x) to be true at any node whose instruction starts with \%x = or is \text{return}, and \text{uses}(x) to be true at any node labeled with an instruction containing \%x as instance of \text{expr}. These predicates will be used in the negative to guarantee independence from certain parts of the local state of the program. In addition to these, we will wish to recognize a subset of \text{def} nodes that are limited in their side-effects to altering only a given local variable (and the program counter). In MiniLLVM, we define \text{alters_at_most}(x) to be true at nodes with instructions of the form \%x = \text{op type expr}, expr or \%x = \text{icmp cmp type expr}, expr. This predicate will be used in the positive, as its negative tells us almost nothing about the effect of the node on the program’s state.

With these constructs, we can define a predicate testing for dead code as follows:

\[
is\text{dead_for}(x) \triangleq \\
(\exists n. \text{SATISFIED AT } n (\text{alters_at_most}(x) \land \neg \text{uses}(x)) \land \\
(\neg \exists X \exists n. \text{def}(x) \cup \text{uses}(x)))
\]
Having defined a test for dead code, we may now define dead code elimination:

\[
\text{dead\_code\_elimination}(n, x) \triangleq \text{SATISFIED\_AT } n (\text{is\_dead\_for}(x)) \text{ ; relabel\_node}(n, \text{br})
\]

We can go in the opposite direction by inserting dead code.

\[
\text{dead\_code\_insertion}(n, \ell) \triangleq \text{SATISFIED\_AT } n \text{ stmt}(\text{br}) \text{ ; relabel\_node}(n, \ell) \text{ ; SATISFIED\_AT } n (\text{is\_dead\_for}(x))
\]

Dead code is determined by looking at future instructions for an overwriting instruction. Redundant assignment is a similar concept, except that it looks backward for an identical instruction.

\[
\text{is\_redundant\_assign}(\ell) \triangleq \text{EXISTS } x. \text{ stmt}(\ell) \land \text{ alters\_at\_most}(x) \land (\neg \text{uses}(x)) \land
\left(\neg \exists y. (\text{uses}(y) \land (E \neg \text{stmt}(\ell) \ S \ \text{def}(y)))) \land
\left(\forall \forall (A \neg \text{def}(x) \ S \ \text{stmt}(\ell)))
\right)
\]

Then we may define redundant assignment elimination as follows:

\[
\text{redundant\_assignment\_elimination}(n, \ell) \triangleq \text{SATISFIED\_AT } n (\text{is\_redundant\_assign}(\ell)) \text{ ; relabel\_node}(n, \text{br})
\]

And we may define redundant assignment insertion by

\[
\text{redundant\_assignment\_insertion}(n, \ell) \triangleq \text{SATISFIED\_AT } n \text{ stmt}(\text{br}) \text{ ; relabel\_node}(n, \ell) \text{ ; SATISFIED\_AT } n (\text{is\_redundant\_assign}(\ell))
\]

The insertion transformations can be used as components of general code motion transformations, such as partial redundant assignment elimination and its dual, partial dead code elimination. In the next example, we will see how they can be used in setting a program up for another transformation.

For a bit more diversity, and a more language-specific transformation, let us consider an example of strength reduction: the replacement of a multiplication by an addition. The reduction may be performed when we have
a node $n$ with a variable $x$ being assigned a multiplication of the variable $i$ and constant $c$, and where $n$ is always preceded by another node assigning $i$ the result of adding a constant $k$ to $i$, and each such node is again preceded by a node at which $x$ is again assigned $i$ times $c$. Having found such a node, then we may replace it by assigning $x$ the sum of $x$ and $c_result$ where $c_result = c \times k$.

$$\text{reduce_mult_to_add}(n) \triangleq$$

$$(\exists x \ i \ k \ c_result. \ \text{SATISFIED\_AT} \ n$$

(\forall \%x = \text{mul int} \ %i, c) \land$

$$(\not\exists (x, i)) \land (c_result = c \times k) \land$

$$\left( (\forall \%x = \text{add int} \ %i, k) \land$$

$$\left( (\forall \%x = \text{mul int} \ %i, c) \land \right) \right) \);$

$$\text{relabel\_node}(n, \%x = \text{add int} \ %x, c_result))$$

While the above transformation is safe, it seems of limited utility. However, the pattern described by the condition in this transformation almost exists when $x$ is a variable in a loop being assigned a multiple of the loop index. The pattern does not quite hold in this situation because along the path leading into the loop, there is no reason to expect that $x$ has been defined, let alone assigned the product of $i$ and $k$. This can be remedied by adding such an assignment immediately before the loop entry.
loop_reduce_mult_to_add(n) ≜
EXISTS m.
   (SATISFIED_AT m entry_for_loop_through(n) ;
    EXISTS x i c k c_result.
      (SATISFIED_AT n
       (stmt(\%x = mul int \%i, c) ∧
        (¬is(x, i)) ∧ (c_result = c × k) ∧
        (\\\A\\\V (A(¬def(x)) ∧ (¬def(i))
          S(stmt(\%i = add int \%i, k) ∧
           (node(m)) ∨
            (\\\A\\\V (A(¬def(x)) ∧ (¬def(i))
              S(node(m) ∨ (¬def(x)) ∧ (¬def(i)))))))));
    EXISTS n'. (skip_insert_before_node(m, n') ;
      dead_code_insertion(n', (\%x = mul int \%i, c)) ;
      reduce_mult_to_add(n))))

The examples above depend in varying degrees upon the language of
the generalized control flow graphs being transformed. We were able to
encapsulate that dependency in predicates such as def and uses. For the
proofs of safety of transformations written using these, it suffices to have
assumptions of the guarantees these predicates need to insure about the
dynamic behavior of the underlying programs, and subsequent proofs for
each individual language using the transformation that it implements these
predicates in a way that satisfies these guarantees. However, for use with
the executable semantics, we must have a way of being able to calculate the
truth of these predicates from the given graphs, and this calls for language
specific definitions before such transformations can be tested.

Because we have parameterized Morpheus by the underlying language
and language-specific predicates, transformations that are safe for a language
are also safe for any extended language into which the original language is
embedded. An interesting subcase of this is extending an existing language
with nodes that have no semantics and are used only as markers in the
graph to add annotations with information from previous transformations.
For example, in loop peeling, we restricted ourselves to loops with a fairly
simple structure to avoid the problem of making multiple copies and even
unwinding subloops forever. If we expand our language to have a copied
instruction, with two out-edges, one labeled old and the other new, then,
when we copy a node, we may add a copied node pointing to each of the original and copied nodes, having first checked that no such node already exists. This ability to mark graph is particularly useful in conjunction with the expression condition language allowing for checking for the existence of such markings, even at a distance in the graph.

6.4. Redundant Store Elimination

We conclude this section with the case study of a redundant store elimination optimization (RSE), which removes stores that may be overwritten before they are used, as shown in Figure 1. Note that the redundant store is replaced by an is_pointer instruction, rather than being eliminated entirely, to ensure that crashes are not delayed in bad executions in which \( e_2 \) is not a pointer-valued expression.

The action involved is simple: replace the instruction at the chosen node with the is_pointer instruction. The side condition should require that there is a node \( n \) containing the store to be eliminated, and that along all paths forward from \( n \) another store occurs that makes \( n \) redundant. To make the optimization safe, we must also require some property to hold on the instructions between \( n \) and the stores that make it redundant; then the Morpheus specification of RSE can be written as:

\[
RSE(\varphi) \triangleq \exists n \ \epsilon_2. \ \text{SATISFIED AT} \ n \ (\text{stmt}(\text{store} \ ty_1 \ \epsilon_1, \ ty_2^* \ \epsilon_2) \ \land \\
\mathcal{A} \ \varphi \ \mathcal{U} \ (\neg \text{node}(n) \ \land \ \text{stmt}(\text{store} \ ty_1' \ \epsilon_1', \ ty_2'^* \ \epsilon_2'))); \ \text{relabel_node}(n, \text{is_pointer} \ \epsilon_2)
\]

To finish this definition, we must find a suitable value for \( \varphi \). A more precise form of RSE would involve using alias analysis to determine whether
memory operations may, must, or cannot refer to the location indicated by $e_2$ at $n$; we could express this precise transformation in Morpheus by adding a predicate that performs this alias analysis (either encoded in FOCTL or provided externally). For the purposes of our example, we instead give a conservative approximation of the necessary condition, one that guarantees the safety of the transformation but may miss some redundant stores. First, we need to require that the value of $e_2$ is not changed, so that we know that successive stores to $e_2$ do indeed overwrite the store at $n$; we can guarantee this through the use of a defined def predicate describing all the instructions that might redefine a variable (recall that MiniLLVM expressions are either constants, or local or global variables). We also need to place some restriction on the kinds of memory operations that can be performed between $n$ and a following store; after all, if the value stored to $e_2$ is used in any way before being overwritten, the store is not redundant. In the absence of alias analysis, we must assume that any reference to a memory location could overlap with $e_2$, so our condition must rule out any load instructions between $n$ and a following store. We can define a predicate not_loads such that not_loads($e$) holds when the current statement is not a load from $e$, and then write the remainder of our side condition as $\varphi_1 \triangleq \neg$def($e_2$) $\land \forall e$. not_loads($e$).

(a) A graph with two redundant stores

(b) A graph with redundant stores?

Figure 2: An RSE example

Using our executable semantics, we can run $RSE(\varphi_1)$ on a range of example GCFGs, such as the graph shown in Figure 2a. From the start node $s_1$, the program initializes a local pointer %x, creates an alias to it in %y and publicizes its location in the global variable @a, and then performs a series of stores to shared memory. The trans_sf function gives us two possible
results for $RSE(\varphi_1)$ on this graph, one in which each of $n_1$ and $n_2$ is replaced by an is_pointer instruction (we could also use $RSE(\varphi_1) \ast$ to apply the transformation repeatedly, replacing both $n_1$ and $n_2$). Furthermore, running each of the transformed programs shows that they produce the same results as the original program: 0 at the location of $%x$, the value of $%x$ at $@a$, and 1 at $@b$. Thus far, $\varphi_1$ appears to be a sufficient condition to ensure the correctness of RSE, and this condition is indeed sufficient for single-threaded programs.

However, when we expand our aim to parallel programs, a potential error becomes apparent. Consider the GCFG in Figure 2b, which has two subgraphs with start nodes $s_1$ and $s_2$. Although the program is not well synchronized, we can see that the false branch in subgraph 2 will never be taken, since if we successfully read the value at $@b$ into $%w$, a value greater than or equal to 1 will have already been stored to $%x$. However, if the store at $n_2$ is removed, then we may reach a state in which $%w$ is 1 and $%v$ is 0, allowing the value 8 to be stored in $@c$. This means that $RSE(\varphi_1)$ will introduce new observable behaviors in the GCFG: in the original graph the final value of the global variable $@c$ is always 7, but in the transformed graph it may be 8. Correct optimizations may rule out some executions (for instance, by optimizing away an outcome of a race condition), but they should never introduce new behavior. Thus, this test case shows that we need to tighten the condition on our RSE optimization to make it safe in a multithreaded environment.

The simplest refinement is to disallow any changes to shared memory between a store to be removed and its following stores. In the example above, if the store to $@b$ in subgraph 1 did not exist, then it would be impossible for subgraph 2 to distinguish between the case in which the store at $n_2$ was removed and the one in which it had already been overwritten by the final store to $%x$. Since we have already ruled out load instructions, we need only prohibit store instructions as well; the appropriate side condition in Morpheus can be written as $\varphi_2 \triangleq \neg \text{def}(e_2) \land ((\forall e. \text{not_loads}(e) \land \text{not_stores}(e)) \lor \text{node}(n))$, where we add a special case to allow for the possibility of looping back through $n$ before reaching the following store. Running trans_sf on $RSE(\varphi_2)$ will then remove the store at $n_1$, but leave $n_2$ untouched. We can run the resulting program and see that, as desired, the transformed program will never produce a value of 8 in $@c$. Through the process of iterated testing and refinement, we have produced an apparently correct form of the RSE optimization on parallel programs—although, if later tests show $\varphi_2$ to be
insufficient to ensure correctness, we can repeat the process and devise a still stronger condition.

7. Related Work

Our work builds on the TRANS approach of expressing optimizations as rewrites on control flow graphs with temporal logic side conditions due to Lacey et al. [2] and Kalvala et al. [3], and the extension of that work to PTRANS due to Mansky and Gunter [12, 13, 14]. We differ from these in the use of a basic action language that respects the structure of control flow graphs, and in the use of a strategy language with explicit scoping of variables and with the structure of a Kleene algebra.

The most closely related system is Cobalt [15], which builds on the same approach. Cobalt optimizations are both executable and automatically verified, though it provides no support for iterative refinement of possibly incorrect specifications. Automation also comes at the cost of expressiveness: Cobalt is limited to a much smaller set of CTL side conditions than TRANS or Morpheus and cannot express optimizations that change the shape of the CFG. Several more expressive systems have been developed along the lines of Cobalt, including Rhodium [16], PEC [17], and XCert [18], but these systems abandon CTL in favor of explicitly manipulating dataflow facts, gaining expressiveness but losing the clean logical relationship between side conditions and graphs.

A wide range of languages have been proposed for specifying rewrites on program graphs. Some systems are tied to a particular target language, while others could be applied to CFGs in any language. Optimix [19, 20] is a graph rewriting system that begins with a very simple set of conditions, and adds new nodes and edges to the graph to track the results of analyses and build up more complex conditions. Via the use of target language extensions, Morpheus can provide similar capabilities, while maintaining the expressiveness of its condition language/ Gospel [21] is another system for graph rewriting with complex side conditions, in which conditions are first-order formulae on data and control flow dependencies. Hoopl [22] is a system for expressing dataflow analyses and compiler transformations in Haskell, based on the work of Lerner et al. [23], in which optimizations are defined in terms of combined (interleaved) steps of analysis and transformation. While these and other systems have a range of basic transformations and condition languages, none share Morpheus’s approach of building from a core set of CFG-preserving rewrites,
which we believe simplifies the process of composing small, reusable, correct components into complex transformations.

In high-level and functional languages, programs are often seen less as structured graphs of instructions and more as syntax trees, in which the nodes are labeled with pieces of syntax and the children of a node are again terms. For efficient implementation, identical subterms may be shared between distinct terms, turning syntax trees into syntax graphs; this gives rise to the field of term graph rewriting. Lean [24] is an early language for describing rewrites on term graphs, and later systems include ... The conditions in these systems are graph patterns, characterizations of the local structure of the subgraph to be rewritten, rather than conditions on potentially lengthy paths from the nodes involved. Edges are not explicitly referenced in these systems, and edge labels do not exist, so there is no correlate to the move_edge action of Morpheus. Finding ways to efficiently identify subgraphs that match a given pattern is a field of research unto itself; GrGen [25] is one example of a system that uses heuristics for fast graph unification and rewriting.

Just as graphs appear in all areas of computer science (and many other fields), graph rewriting has arisen as a problem in various other areas. A few applications include rewriting dependency graphs to propagate changes in large software systems [26], rewriting communication graphs to give semantics to distributed systems [27], and transforming software engineering models into different modeling languages [28]. In all of these systems, as in Morpheus, the transformation language breaks down into three main components: a language of basic graph modifications, a language for specifying conditions, and a language for composing rewrites into more complex transformations. PROGRES [29] stands out as a graph transformation language with a visual interface; because of the fundamentally visual nature of graphs, a visual approach to representing transformations improves usability significantly. (Should we say something about wanting a visual interface for Morpheus?) In nearly all of these applications, the condition language gives a way of specifying subgraph patterns rather than temporal-logic-style whole-graph properties; the need for conditions over extended paths as in “not used until redefined” seems to be specific to the domain of compiler optimizations.

CompCert [30], the definitive example of a proof of compiler correctness, also includes a Coq-based framework for specifying and verifying compiler optimizations; executable semantics are obtained by extracting code from the Coq definition, guaranteeing its correctness. Their specifications follow the traditional algorithmic approach to dataflow analysis, with the conditions
under which an optimization should be applied expressed as a set of transfer functions for dataflow equations. The ongoing CompCertTSO project [31] seeks to add support for concurrency to CompCert, and has involved the specification and verification of a small number of concurrency-specific optimizations, as well as a range of sequential optimizations that can be lifted as-is to the concurrent case.

Jifeng and Bowen [32] have also developed a language for specifying and prototyping compiler transformations, focusing particularly on the code generation phase of compilation. Their language consists of if-expressions, analogous to sequencing conditions and actions in Morpheus, and is implemented as a set of Horn clauses in Prolog, thus having no explicit scope. Rather than giving operational semantics for a real-world target language, they model programs directly as sequences of modifications to the machine state. Because they deal primarily with language-to-language translation, their transformations are not innately composable, and they deal largely with local peephole optimizations rather than those involving dataflow analysis.

8. Conclusion and Future Work

In this paper, we show the use of the Morpheus specification language in designing and prototyping compiler optimizations in terms of graph transformations. By expressing optimizations as rewrites on control flow graphs with temporal logic side conditions, Morpheus allows for a more direct expression of the logic behind transformations. The mathematical semantics of Morpheus are accompanied by an executable semantics, allowing us to run Morpheus specifications directly on program graphs. The executable semantics relies on an algorithm for finding satisfying models of first-order CTL formulae on a graph, which in combination with an SMT solver can efficiently find all possible locations at which a transformation applies. Because the basic actions of Morpheus each individually preserve the well-formedness of control flow graphs, we can easily analyze complex transformations as compositions of simple, innately safe components. With its combination of abstract and executable semantics and its focus on compositional reasoning, Morpheus lays the groundwork for a unified platform for specifying, testing, and verifying optimizations.

While we have implemented the executable semantics of Morpheus in F# with Z3 integration, we are also interested in developing it in the K Framework [33] for programming language specification. A K implementation of
Morpheus could take advantage of built-in state-space search functionality, as well as the wide range of languages that have been given formal semantics in K, including C, OCaml, and a more complete version of LLVM. We also intend to move forward with the formal verification of optimizations specified in Morpheus in the Isabelle theorem prover [34], and ultimately hope to link our executable semantics with our abstract semantics through a formal soundness proof.

Acknowledgements. This material is based upon work supported in part by NSF Grant CCF 13-18191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

We would like to thank Elizabeth A. Kelly, Blake A. Petermeier, Grant J. Czajkowski, Maria Kotsifakou, Taiyu Dong, Wenjie Zhu, Michael Bay, and Terence Nip, who contributed to the development of many of the specifications of transformations in the paper and their proofs of safety in an earlier version of this system specified in the theorem prover Isabelle.


URL http://doi.acm.org/10.1145/1863523.1863539

URL http://doi.acm.org/10.1145/503272.503298


URL http://dx.doi.org/10.1007/11841883_27

URL http://dl.acm.org/citation.cfm?id=645545.656039

URL http://doi.acm.org/10.1145/23005.24038


[34] L. C. Paulson, Isabelle: The next 700 theorem provers, CoRR cs.LO/9301106.
URL http://arxiv.org/abs/cs.LO/9301106