Guaranteed Optimization for Domain-Specific Programming

Todd L. Veldhuizen

Indiana University, Bloomington, Indiana 47401 USA
tveldhui@acm.org

Abstract. For software engineering reasons, it is often best to provide domain-specific programming environments in the context of a general-purpose language. In our view general-purpose languages are not yet general-purpose enough, and progress needs to be made before we can provide domain-specific languages that are both fast and safe. We outline some goals in this regard, and describe a possible implementation technology: *guaranteed optimization*, a technique for building compilers that provide proven guarantees of what optimizations they perform. Such optimizers can provide capabilities similar to staged languages, and thus provide the relevant performance improvements. They can also function as decision procedures, suggesting an approach of 'optimizers as theorem provers,' in which optimizing compilers can be used to check domain-specific safety properties and check proofs embedded in programs.

1 Introduction

There are several competing strategies for providing domain-specific programming environments: one can construct a wholly new language, extend an existing language, or work within an existing language. New languages are appealing because they're fun to design and allow radical departures in syntax and semantics from existing languages; an example is logic-based program synthesis (e.g., [1]), in which purely declarative languages act as specifications for automatic programming. For non-programmers, a simple language that does exactly what they need can be less intimidating than general-purpose languages. However, new languages have drawbacks, too:

– New languages are hard to get right — semantics is a surprisingly tricky business.
– Users have to learn a new programming language, which can discourage adoption.
– Compilers require ongoing support to keep up with changing operating systems and architectures. One-off languages requiring special compilers are often research projects that founder when students graduate and professors move on to other interests.
– You can't use features of multiple DSLs in one source file. For example, there currently exists a Fortran-like DSL that provides sparse arrays, and another

that provides interval arithmetic. However, if one wants both sparse arrays and intervals, there is no compiler that supports both at once.

- One-off languages bring with them software-engineering risks, since they tend to lack the tool support (optimizing compilers, debuggers, development environments) of mainstream languages. To go down this road is to expose programmers to the prospect of a hundred thousand lines of misbehaving legacy code in an obscure language with no debugger.

Perhaps most importantly, one-off languages tend to lack many of the features one finds in mainstream languages, such as exceptions, language interoperability, threads, GUIs, and general-purpose I/O. Lacking these amenities, they can either languish in a limited niche, or their users will recreate what's missing to make them more generally useful. There is a connection to be drawn between the feature creep that has turned Matlab, Perl, and Fortran into general-purpose languages and the process of creolization in natural languages. In creolization, pidgin languages developed for communication between two cultures (and here we might view Matlab as a "pidgin" of linear algebra notation and Basic) acquire native speakers and enlarge to become full-fledged languages. If people are taught a limited language, they will of necessity make it big enough to express everything that needs saying — no matter how awkwardly. In the New Guinea pidgin language Tok Pisin, which lacks the word *moustache*, the phrase *gras bilong maus* (grass belong mouth) conveys a visual impression of grass growing from someone's lip. Such circumlocutions are akin to idioms developed by programming language communities to emulate missing language features. For example recursion, modules and dynamic memory management are all features absent from Fortran 77 and awkwardly emulated by users. These features were added to the language by Fortran 90. Instead of introducing a wholly new language which, if it becomes popular, is bound to evolve haphazardly toward general-purposeness, it makes sense to start with a general-purpose language powerful enough to encompass domain-specific uses.

So: unless one truly needs a language with radically new semantics or syntax, it's prudent to work with a well-supported, general-purpose language (cf. [2]). Unfortunately the existing mainstream languages aren't extensible enough to do interesting, high-performance DSL work in a natural way. Therefore a fruitful direction for research is to extend the reach of general-purpose languages; in this we pursue the old, probably unattainable dream of a universal language suitable for all purposes, a dream that goes back at least to Leibniz and his grandiose vision of a *Characteristica Universalis* in which to formalize all human thought (cf. [3]). In the past half-century useful connections have been found between logic, type theory, algebra, compilers, and verification; perhaps it is a good time to examine these threads of convergence and take stock of the prospects for universal programming languages.

Structure of this paper. This is a paper in two parts. In the first we describe hopes for general-purpose languages, which can be summarized as "pretty, fast, safe." This provides background for the second half of the paper in which we propose technologies to make languages more general-purpose: guaranteed optimization, which addresses some aspects of performance, and proof embeddings, which address safety.

2 Pretty, Fast, Safe: Meditations on Extensible Languages

Aiming for universality means we should seek to span the gamut of needs, rather than targeting specific niches. Thus we begin with a broad survey of many different niches, setting our sights on one language to cover them all.

Our wish-list for a universal language can be summarized under these broad headings:

– **Pretty**. Programmers ought to be able to describe their problem in a form appropriate to their domain. We know that specialists invent new jargon and notations for their problem domains, and indeed that the development of notation is often central to gaining insight.[1]
– **Fast**. Performance should approach that of hand-crafted, low-level code, even for code written at high levels of abstraction.
– **Safe**. There should be mechanisms to statically check domain-specific safety properties.

In this paper we are concerned mostly with the *fast* and *safe* aspects, the pretty part — extensible syntax — having been studied in depth for decades, with macro systems, extensible parsers and the like. The problem of how to provide extensible languages with performance and safety, though, is a bit less well-explored.

Fast. With current compilers there is often a tradeoff between the expressiveness of code and its performance: code written close to the machine model will perform well, whereas code written at a high level of abstraction often performs poorly. This loss in performance associated with writing high-level code is often called the *abstraction penalty* [4–6]. The abstraction penalty can arise both from using a high level of abstraction, and also from syntax extensions (e.g., naive macro expansion). For this reason, compilers for *fast* extensible languages need to minimize this abstraction penalty. Traditional compiler optimizations are the obvious solution: disaggregation, virtual function elimination, and the like can greatly reduce the abstraction penalty. One of the shortcomings of current optimizing compilers is that they are unpredictable, with the result that performance-tuning is a fickle and frustrating art rather than a science. In the second half of this paper we propose a possible solution: optimizers that provide proven guarantees of what optimizations they will perform, thus making performance more predictable.

[1] "Some of the greatest advances in mathematics have been due to the invention of symbols, which it afterwards became necessary to explain; from the minus sign proceeded the whole theory of negative quantities." – Aldous Huxley

Beyond reducing the abstraction penalty, many problem areas admit *domain-specific optimizations* which can be exploited to achieve high performance. For example, operations on dense arrays admit an astonishing variety of performance improvements (e.g., [7]). A fundamental issue is whether domain-specific optimization is achieved by *transformation* or *generation* of code:

– In transformation-based optimization one has low-level code (e.g., loop nests) that may represent high-level abstractions (e.g., array operations), and the optimizer seeks to recognize certain inefficient patterns and replace them with efficient code. Two difficulties with this approach are the *recognition problem* of finding pattern matches, and the fact that many interesting optimizations may violate the compiler's notion of behavioural equivalence (thus, a good transformation may require one to play fast-and-loose with semantics, not a good thing.)
– In *generative optimization*, one starts from a high-level operation such as $A = B + C * D$ and generates efficient code for it. This avoids both the recognition problem and taking liberties with semantics.

The transformation-based approach can be thought of as naive translation followed by smart optimization; conversely, the generative approach can be viewed as smart translation.

It's not practical to include all such domain-specific optimizations in compilers, for economic reasons [8, 9]. Better, then, to package domain-specific optimizations with libraries; this is the basic idea behind *active libraries* [10]. Such libraries can capture performance-tuning expertise, providing better performance for average users, while still allowing experts to "roll-their-own." This idea is similar in spirit to ongoing work by the parallel skeletons community (e.g., [11, 12]) who seek to capture patterns of parallel computing in reusable libraries of "skeletons." There are many approaches to packaging optimizations and libraries together:

– "Meta-level" or "open compiler" systems let users manipulate syntax trees and other compiler data structures (e.g., [13–16]). While this technique provides enormous raw power, it does have drawbacks: obviously, letting users change compiler data structures raises some safety issues. Also, mainstream languages tend to be rather big, syntactically speaking, and so manipulating their syntax trees brings with it a certain complexity. It's unclear how difficult this approach might be to implement for industrial languages since there are often many compilers for a given language, each having its own internal representations, and so one gets into messy issues of politics and economics.
– Annotations (e.g., [17]) can provide additional hints to the compiler about semantics and permissible transformations. While this approach is more limited in scope than metalevel approaches, it does avoid the complexity of manipulating syntax trees directly.
– Rewriting (e.g., [18–20]) is a promising technology for capturing domain-specific code transformations; it can express simple transforms elegantly.

It's not clear yet how cleanly one can express transforms that require some mixture of analysis and code generation.

– Staging (e.g., [21, 22]) and partial evaluation (e.g., [23]) lie somewhere between meta-level approaches and annotations in terms of power; they avoid the complexity of manipulating syntax trees while still allowing very useful optimizations. Staging has been used to great effect in C++ to provide high-performance libraries for scientific computing, which we'll discuss shortly. In particular, staging and partial evaluation allow one to specialize code and (more generally) do component generation, which can have an enormous benefit to performance.

Safe. There are diverse (and firmly held!) ideas about what it means for a program to be safe. The question is what makes programs safe *enough*, absolute safety being unattainable in practice. The pragmatic approach is to recognize the existence a wide spectrum of safety levels, and that users should be free to choose a level appropriate to their purpose, avoiding a "one size fits all" mentality. At the low end of the spectrum are languages such as Matlab that do almost no static checking, deferring even syntactic checking of functions until they are invoked. At the other end is full-blown deductive verification that aims to prove correctness with respect to a specification. To aim for universality, one would like a single language capable of spanning this spectrum, and in particular to allow different levels of safety for different aspects of a program. For instance, it is common to check type-correctness statically, while deferring array bounds-checking until run time; this clearly represents two different standards of safety. Even program verification is typically applied only to some critical properties of a program, rather than attempting to exhaustively verify every aspect. Thus it is important to allow a mixture of safety levels within a single program.

Safety checks may be dynamic, as in (say) Scheme type checks [24] or Eiffel pre- and post-condition checking [25]. Dynamic checking can detect the presence of bugs at run-time; static checking can prove the absence of some bugs at compile-time. A notable difference between the two is that fully automatic static checking must necessarily reject some correct programs, due to the undecidability of most nontrivial safety conditions. Within static checking there is a wide range of ambition, from opportunistic bug-finding as in LCLint [26] and Metal [27], to lightweight verification of selected properties as in extended static checking [28] or SLAM [29], to full-blown deductive verification (e.g., PVS [30], Z [31], VDM [32]).

Systems that check programs using static analysis can be distinguished on the style of analysis performed: whether it is flow-, path-, or context-sensitive, for example. A recent trend has been to check static safety properties by shoehorning them into the type system, for example checking locks [33, 34], array bounds [35], and security properties [36]. Type systems are generally flow- and context-invariant, whereas many of the static safety properties one would like to check are not. Thus it's not clear if type-based analyses are really the best way to go, since the approximations one gets are probably so coarse as to be of limited use in practice.

We can also distinguish between approaches in which safety checks are external to the program (as annotations, additional specifications, etc.) versus approaches in which safety checks are part of the program code (for example, run-time assertions, pre- and post-condition checking). Any artifact maintained separately from the source code will tend to diverge from it, whether it be documentation, models or proofs. The "one-source" principle of software engineering suggests that safety checks should be integrated with the source code rather than separate from it. Some systems achieve this by placing annotations in the source code (e.g., ESC/Java [37]). However, this approach does not appear to integrate easily with staging; for example, can a stage produce customized safety checks for a later stage, when such checks are embedded in comments? Comments are not usually given staging semantics, so it's unclear how this would work. Having safety checks *part of the language* ensures that they will interact with other language features (e.g., staging) in a sensible way.

Rather than relying solely on "external" tools such as model checkers or verification systems, we'd like as much as possible to have safety checking integrated with compilation. Why? First, for expedience: many checking tools rely on the same analyses required by optimization (points-to, alias, congruence), so it makes sense to combine these efforts. But our primary reason is that by integrating safety checking with compilers, we can provide libraries with the ability to perform their own static checks and emit customized diagnostics.

The approach we advocate is to define a general-purpose safety-checking system which subsumes both type-checking and domain-specific safety checks. Thus types are not given any special treatment, but rather treated the same as any other "domain-specific" safety property. Ideally one would also have extensible type systems, since many abstractions for problem domains have their own typing requirements. For example, deciding whether a tensor expression has a meaningful interpretation requires a careful analysis of indices and ranks. In scientific computing, dimension types (e.g., [38]) have been used to avoid mistakes such as assigning meters-per-second quantities to miles-per-hour variables. Having type checking handled by a general-purpose safety-checking system will likely open a way to extensible type systems.

Yet another aspect of safety checking is whether it is fully automatic (static analysis, model checking) or semi-automatic (e.g., proof assistants). There are many intriguing uses for a compiler supporting semi-automatic checking. By this we mean that a certain level of automated theorem proving takes place, but when checking fails, users can provide supplemental proofs of a property in order to proceed. This opens the way for libraries to issue *proof obligations* that must be satisfied by users (for example, to remove bound checks on arrays). Interesting safety properties are undecidable; this means any safety check must necessarily reject some safe programs. Thus, proof obligations would let users go beyond the limits of the compiler's ability to automatically decide safety.

2.1 Reminiscences of a C++ apostate.

C++ has enjoyed success in providing domain-specific programming environments, and deserves attention as a case study. What distinguishes it is its staging mechanism (template metaprogramming), which has made possible the generation of highly efficient implementations for domain-specific abstractions. The key to this success is C++'s template mechanism, originally introduced to provide parameterized types: one can create template classes such as $\mathsf{List}\langle T\rangle$ where T is a type parameter, and instantiate it to particular instances such as $\mathsf{List}\langle\mathsf{int}\rangle$ and $\mathsf{List}\langle\mathsf{string}\rangle$. (This idea was not new to C++ — a similar mechanism existed in Ada, and of course parametric polymorphism is a near cousin.) This instantiation involves duplicating the code and replacing the template parameters with their argument values — similar to polyvariant specialization in partial evaluation (c.f. [39]). In the development of templates it became clear that allowing dependent types such as $\mathsf{Vector}\langle 3\rangle$ would be useful (in C++ terminology called *non-type* template parameters); to type-check such classes it became necessary to evaluate expressions inside the $\langle\rangle$ brackets, so that $\mathsf{Vector}\langle 1+2\rangle$ is understood to be the same as $\mathsf{Vector}\langle 3\rangle$. The addition of this evaluation step turned C++ into a staged language: arbitrary computations could be encoded as $\langle\rangle$-expressions, which are guaranteed to be evaluated at compile-time. This capability was the basis of template metaprogramming and expression templates. A taste for these techniques is given by this definition of a function pow to calculate x^n (attribution unknown):

```
template<unsigned int N>
inline float pow(float x)
{ return pow<N % 2>(x) * pow<N / 2>(x*x); }

template<> inline float pow<1>(float x) { return x; }
template<> inline float pow<0>(float x) { return 1; }
```

The code `y=pow<5>(x)` expands at compile-time to something like `t1=x*x; t2=t1*t1; y=t2*x`. Using such techniques, C++ has been used to provide high-performance domain-specific libraries, for example POOMA [40] and Blitz [41] for dense arrays, MTL [42] and GMCL [43] for linear algebra. Blitz is able to do many of the dense array optimizations traditionally performed by compilers, such as loop fusion, interchange, and tiling. POOMA generates complicated parallel message-passing implementations of array operations from simple user code such as "A=B+C*D." GMCL does elaborate component generation at compile-time, producing a concrete matrix type from a specification of element type, sparse or dense, storage format, and bounds checking provided by the user. Thus C++ is interesting because these libraries have been able to provide capabilities previously provided by optimizing compilers or component generation systems.

For syntactic abstraction, C++ provides only a fixed set of overloadable operators, each with a fixed precedence. This makes it a rather poor cousin of custom infix operators or general macro systems. However, people get surprising mileage out of this limited capability: one of the lessons learned from C++ is that the combination of staging and overloadable operators can be parlayed into customized parsing by turning a string of tokens into a tree and then traversing the tree in the compile-time stage to transform or 'compile' the expression.

Another useful lesson from the C++ experience is that staging can be used to provide static safety checks. Examples of this include 'concept checking' in C++ [44, 45], which essentially retrofits C++ with bounded polymorphism; SIUnits [46], which realizes the idea of dimension types [38] in C++ to check type safety with respect to physical units; also, MTL and Blitz check conformance of matrices and arrays at compile-time. And as a more general example, we point to the `ctassert<>` template [47] which provides a compile-time analogue of dynamic `assert()` statements.

C++ was not intended to provide these features; they are largely serendipitous, the result of a flexible, general-purpose language design. Even now — a good decade after the basic features of C++ were laid down — people are still discovering novel (and useful!) techniques to accomplish things that were previously believed impossible; just recently it was discovered that compile-time reflection of arbitrary type properties was possible [48], something previously thought impossible. The lesson to be taken is that programming languages can have emergent properties; we mean "emergent properties" in the vague sense of surprising capabilities whose existence could not be foretold from the basic rules of the language. And here we must acknowledge Scheme, a small language that has proven amazingly capable due to such emergent properties. Such languages have an exuberance that makes them both fun and powerful; unanticipated features are bursting out all over! That emergent properties have proven so useful suggests that we try to foster languages with such potential. In the second half of this paper we describe our efforts in this direction.

3 Guaranteed Optimization and Proof Embeddings

The central technology in our approach to providing extensible languages is *guaranteed optimization*, which can be used to reduce abstraction penalty, remove overhead from naive expansion of syntax extensions, and provide staging-like capabilities. In the latter parts of this section we describe how guaranteed optimization can be used to prove theorems and check proofs embedded in programs.

3.1 Optimizing programs thoroughly and predictably

Guaranteed Optimization is a method for designing compilers that have proven guarantees of what optimizations they will perform. Here we give a taste for the method and explore its application to domain-specific programming environments; for a detailed explanation see [49, 50].

Guaranteed optimization is a "design-by-proof" technique: by attempting to prove a compiler has a certain property one uncovers failures in its design, and when at last the proof succeeds the compiler has the desired property. The inspiration comes from normal forms. Ideal optimizing compilers would compute normal forms of programs: every program would be reduced to some "optimal" program in its semantic equivalence class. This goal is of course out of reach, since program equivalence is undecidable; although we might be able to produce "optimal" programs for some classes of programs, we can't do so in general for all programs. The best we can hope for is compilers that find normal forms of programs within some large, decidable subtheory of program equivalence, and this is what guaranteed optimization does.

A starting point is the goal that optimizing compilers should undo transformations we might apply to programs to make them "more abstract" for software-engineering purposes, for example replacing "$1 + 2$" with

```
x = new Integer(1);
y = new Integer(2);
x.plus(y).intValue();
```

Not that anyone would write *that*, exactly, but similar code is routinely written for encapsulation and modularity. We can represent such transformations by rewrite rules. Some trivial examples of rewrites are:

$$\begin{array}{ll} R1. & x \to x + 0 \\ R2. & x \to car(cons(x, y)) \\ R3. & x \to \mathsf{if\ true\ then}\ x\ \mathsf{else\ y} \\ & \quad\vdots \end{array}$$

These rules are, of course, completely backward from the usual approach: we work with rules that *de-optimize* a program. The obvious question is: why not devise an optimizer by reversing the direction of the arrows, using (say) Knuth-Bendix completion? The answer is that we work with an infinite number of such rules, some of which are conditional; thus we violate two requirements of Knuth-Bendix (which applies to a finite number of unconditional axioms.) Moreover, reversing the direction of the arrows can turn unconditional rewrites into conditional rewrites with undecidable conditions; and sensible approximations to the conditions require global analysis.

It turns out that the unusual approach of considering de-optimizing rules leads to a usable proof technique: we can prove that certain compilers undo any sequence of rule applications *in a single application of the optimizer*, yielding a program that is "minimally abstract." Figure 1 illustrates.

In the next section we sketch the proof technique to give a taste for the methodology and results; for a detailed description see [49, 50].

An overview of the proof technique In what follows, we write $\to$ for the "de-optimizing" rewrite relation and $x \leftrightarrow y$ ("convertible") if $x \to y$ or $y \to x$. We write $\xrightarrow{*}$ and $\overset{*}{\leftrightarrow}$ for the reflexive, transitive closure of $\to$ and $\leftrightarrow$, respectively.

The proof technique applies to optimizers based on the *superanalysis* approach [51, 52]: instead of separate optimizing passes, there is instead a single, combined analysis followed by one transformation step. Superanalysis avoids the phase ordering problem of multiple optimization passes, a property crucial to the

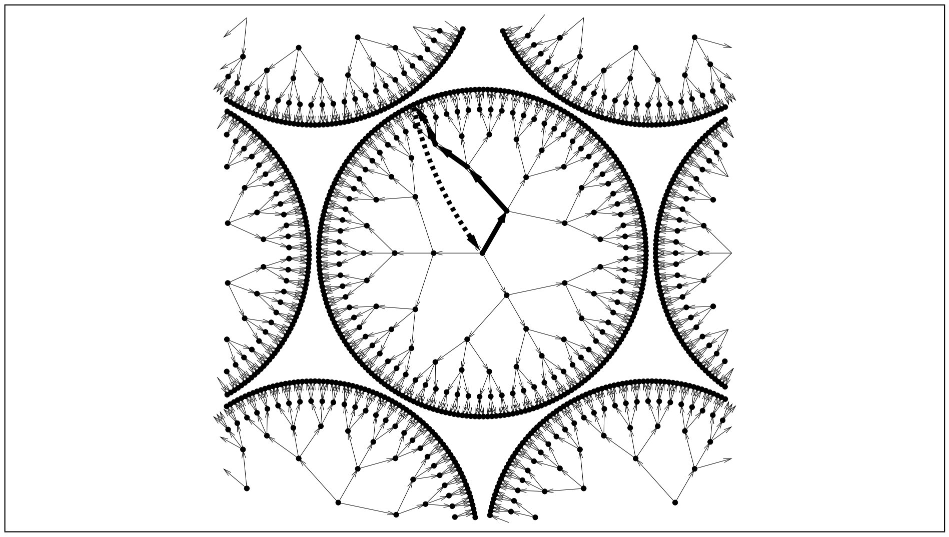
Fig. 1: A simplified sketch of Guaranteed Optimization. Each point represents a program and arrows indicate de-optimizing rewrites. The optimizer guarantees that it will undo any sequence of de-optimizing rewrites (bold arrows) in a single step (dashed arrow). Each circle represents a "behavioural equivalence class" of programs judged to have the same behaviour by the optimizer.

proof of guaranteed optimization. (As we discuss later, superanalysis also lets optimizers function as effective decision procedures in the presence of circularity e.g. recursion). Superanalysis-based optimizers consist of three steps:

1. The program text is examined to produce a system of analysis equations;
2. These equations are solved to a fixpoint;
3. Using this fixpoint solution, the program is transformed.

We can represent this process graphically for some program p as:

$$\begin{tikzcd} p \arrow[r, "(1)"] & \text{analysis equations} \arrow[r, "(2)"] & \text{solution} \arrow[r, "(3)"] & \text{transformed program} \end{tikzcd}$$

where the numbers (1), (2), (3) refer to the steps above. The essence of the proof technique is to consider a de-optimizing rewrite $p \rightarrow p'$, and compare what happens to both p and p' in each step of the optimizer:

$$\begin{tikzcd} p \arrow[r, "(1)"] \arrow[d, "\text{rewrite}"'] & \text{analysis equations} \arrow[r, "(2)"] \arrow[d, dotted] & \text{solution} \arrow[r, "(3)"] & \text{transformed program} \arrow[d, equal] \\ p' \arrow[r, "(1)"] & \text{analysis equations} \arrow[r, "(2)"] & \text{solution} \arrow[r, "(3)"] & \text{transformed program} \end{tikzcd}$$

If the analysis is compositional — which for the optimizations we consider, it is — then a rewrite $p \to p'$ can be viewed as inducing a rewrite on the analysis equations, shown above by a dotted line. (For example if one replaces x by $x + 0$ in a program, there will be corresponding changes in the analysis equations to add new equations for 0 and $x + 0$.) By reasoning about this "induced rewrite" on the analysis equations one can prove properties of the solution that imply the transformed programs are equal (hence the double vertical line in the above figure.) It is convenient to think of the rewrite in terms of its context and redex: in a rewrite $x{+}y \;\to\; x{+}(y{+}0)$ we have the context $x{+}[\,]$ and the redex y, where $[\,]$ denotes a *hole*. Thus the context is the portion of the program unchanged by the rewrite.

For each rewrite rule $R1, R2, \ldots$ one proves a lemma, a simplified example of which we give here for some rule $R1$:

Lemma 1 *If $p \to p'$ by rule $R1$, then (i) the analysis equations for p' have the same solution as the equations of p for program points in the rewrite context, and (ii) the transformed version of p' is the same as the transformed version of p.*

One proves a lemma of the above form for each rewrite considered. The effect of a rewrite on the fixpoint solution of a system of equations is reasoned about using the theory of fixpoint-preserving transformations and bisimilarity [53, 54]. We use these techniques to prove that analysis equations for program points in the context of p are bisimilar to those in p', and thus the solution is unchanged for analysis variables in the context; and we prove enough about the solution to the p' equations to show that the additional code introduced by the rewrite is removed by the transformation step.

Proving these lemmas requires adjusting the design of the analysis and transformation so that the proof can succeed; thus, the proof technique is really a *design* technique for optimizing compilers. Once the lemmas are established, we can show that any sequence of rewrites is undone by the optimizer by a straightforward induction over rewrites, which we illustrate by stacking the above diagrams (Figure 2). Thus for any sequence of rewrites $p \to p' \to p'' \to \ldots$, we have that the transformed versions of p, p', p'', $\ldots$ are all equal. Writing $\mathcal{O} : \mathsf{Program} \to \mathsf{Program}$ for the optimizer, a guaranteed optimization proof culminates in a theorem of the form:

Theorem 1 (Guaranteed Optimization) *If $p \xrightarrow{*} p'$, then $\mathcal{O}p = \mathcal{O}p'$.*

This theorem simply states that any sequence of "de-optimizing" rewrites is undone by the optimizer. This theorem has some interesting implications. Recall that the kernel of $\mathcal{O}$ is $\ker \mathcal{O} = \{(p_1, p_2) \mid \mathcal{O}p_1 = \mathcal{O}p_2\}$. If the optimizer is sound, then $(p_1, p_2) \in \ker \mathcal{O}$ implies that p_1 and p_2 are behaviourally equivalent. Thus, one can view the optimizer $\mathcal{O}$ as computing a normal form of programs with respect to a decidable subtheory of behavioural equivalence.

This notion of the kernel of an optimizer is generally useful, since it captures what we might call the "staging power" of a compiler; if $\mathcal{O}_A$ and $\mathcal{O}_B$ are two

$$\begin{tikzcd}
p \arrow[r, "(1)"] \arrow[d, "\text{rewrite}"'] & \text{analysis equations} \arrow[r, "(2)"] \arrow[d, dotted] & \text{solution} \arrow[r, "(3)"] & \text{transformed program} \arrow[d, equal] \\
p' \arrow[r, "(1)"] \arrow[d, "\text{rewrite}"'] & \text{analysis equations} \arrow[r, "(2)"] \arrow[d, dotted] & \text{solution} \arrow[r, "(3)"] & \text{transformed program} \arrow[d, equal] \\
p'' \arrow[r, "(1)"] \arrow[d] & \text{analysis equations} \arrow[r, "(2)"] \arrow[d, dotted] & \text{solution} \arrow[r, "(3)"] & \text{transformed program} \arrow[d, equal] \\
\vdots & \vdots & \vdots & \vdots
\end{tikzcd}$$

Fig. 2: A diagram illustrating induction over rewrites: any sequence of de-optimizing rewrites is undone by the optimizer.

optimizing compilers and ker $\mathcal{O}_A \subseteq$ ker $\mathcal{O}_B$, we can conclude that $\mathcal{O}_B$ is a more powerful optimizer.

Another useful property one can prove with guaranteed optimization is the following: if one defines an *abstraction level* $AL(p)$ of a program p as the length of the longest chain $p_0 \to \ldots \to p$, then by imposing a few further requirements on the optimizer one can attain $AL(\mathcal{O}p) = 0$. Such optimizers find a minimal program with respect to the metric AL. Thus with appropriate "de-optimizing" rules $\to$, guaranteed optimization can address the abstraction penalty: optimized programs are guaranteed to be "minimally abstract" with respect to the rewrites $\to$.

Guaranteed Optimization as staging. Having described the basics of guaranteed optimization, we now examine its application to extensible compilation. In particular, we are interested in using guaranteed optimization to provide similar capabilities to staging and partial evaluation. We start by defining an evaluation relation $\twoheadrightarrow$ sufficient to evaluate a purely functional subset of the intermediate language, for example:

$$\begin{array}{lrl}
E1. & \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \twoheadrightarrow e_1 & \\
E2. & \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \twoheadrightarrow e_2 & \\
E3. & n_1 + n_2 \twoheadrightarrow n_3 & \text{where } n_i \in \mathbb{Z},\ n_3 = n_1 + n_2 \\
 & \vdots &
\end{array}$$

Now, a good optimizer should clearly perform some of these evaluation steps at compile time by constant folding. We can use Guaranteed Optimization to design compilers that fully reduce a program with respect to $\twoheadrightarrow$. Suppose the compiler guarantees that applications of the "de-optimizing" rule $R3$ are undone:

$$R3. \quad x \rightarrow \mathsf{if\ true\ then}\ x\ \mathsf{else\ y}$$

Clearly the right-hand side of rule $R3$ matches the left-hand side of rule $E1$; so the optimizer will fully reduce a program with respect to $E1$. By making the set of "de-optimizing" rewrites $\rightarrow$ big enough, in principle one can prove that:

$$\underset{\text{(evaluation relation)}}{\twoheadrightarrow} \quad \subseteq \quad \underset{\text{(guaranteed optimization)}}{\overset{*}{\leftrightarrow}} \tag{1}$$

This becomes interesting and useful when the evaluation relation $\twoheadrightarrow$ encompasses a Turing-complete subset of the language; then we have functionality closely related to that of staging. In staging (e.g. [21, 22]; see also the chapter by Taha, this volume), one has evaluation relations such as:

$$\langle x + \tilde{}\,(1 + 2)\rangle \quad \twoheadrightarrow \quad \langle x + 3\rangle$$

Guaranteed optimization gives staging-like capability, but with fewer explicit annotations:[2]

$$x + (1 + 2) \quad \twoheadrightarrow \quad x + 3$$

So there is less of a syntactic burden to users, who no longer have to escape and defer obvious things; it is more like partial evaluation in this respect.

To summarize, guaranteed optimization can reduce the abstraction penalty; its staging-like capabilities can be used to remove code introduced by syntax extensions and to perform domain-specific optimizations.

3.2 Optimizers as theorem provers

To provide domain-specific safety checks, we need some kind of theorem proving capability. Many safety checks require the same analyses as optimization — points-to, congruence, and the like; the approach we're exploring is to use the optimizer itself as an theorem-proving engine.

Sound program optimizers can be used to prove theorems: for example, if x is of type *int* and the optimizer replaces $x + 0$ with x, then since the optimizer is sound $x + 0 = x$ must hold in the theory of *int*.

We write $\sim$ for behavioural equivalence on programs: if $p_1 \sim p_2$, then the behaviour of programs p_1 and p_2 are indistinguishable (Our working notion of behavioural equivalence is weak bisimilarity, with the observable behaviour of

[2] To reach Turing-completeness of $\twoheadrightarrow$ in guaranteed optimization, some annotations are still necessary to indicate which function applications are to be unfolded, and one must allow the possibility of nontermination of the optimizer, as in partial evaluation and staging.

a program being its interaction with the operating system kernel and other processes via shared memory [55, 56]).

For the optimizer to be sound, we require that $p \sim \mathcal{O}p$ — a program must be behaviourally equivalent to its optimized version. Therefore $\mathcal{O}p = \mathcal{O}p'$ implies $p \sim p'$, or equivalently:

$$\ker \mathcal{O} \;\subseteq\; \sim \tag{2}$$

Thus one can view program optimizers as deciding a weaker behavioural equivalences on programs. Guaranteed optimization is, in essence, a proof that ker $\mathcal{O}$ satisfies certain closure properties related to the de-optimizing rewrites. From Theorem 1 (Guaranteed Optimization) we have $p \overset{*}{\to} p' \;\Rightarrow\; \mathcal{O}p = \mathcal{O}p'$. This implies $\overset{*}{\leftrightarrow} \;\subseteq \ker \mathcal{O}$; together with soundness of $\mathcal{O}$, we have:

$$\overset{*}{\leftrightarrow} \;\subseteq\; \ker \mathcal{O} \;\subseteq\; \sim \tag{3}$$

The "de-optimizing" rewrites of guaranteed optimization can be viewed as oriented axioms. A compiler for which the guaranteed optimization proof succeeds is an effective decision procedure for the theory generated by these axioms (or, usually, a sound superset of the axioms). In related work [57] we describe a correspondence between the problem of how to effectively combine optimizing passes in a compiler, and how to combine decision procedures in an automated theorem prover. Rewrite-based or *pessimistic* optimizers can decide combinations of inductively defined theories in a manner similar to the Nelson-Oppen method of combining decision procedures [58]. On the other hand, optimizers based on *optimistic superanalysis*, of which guaranteed optimization is an example, can decide combinations of (more powerful) coinductively defined theories such as bisimulation.

This correspondence suggests using the compiler as a theorem prover, since the optimizer can prove properties of run-time values and behaviour. The optimizer fulfills a role similar to that of simplification engines in theorem provers: it can decide simple theorems on its own, for example $x + 3 = 1 + 2 + x$ and $car(cons(x, y)) = x$. A number of interesting research directions are suggested:

– By including a language primitive $\mathsf{check}(\cdot)$ that fails at compile-time if its argument is not provably true, one can provide a simple but crude version of domain-specific static checking, that would, in principle and assuming Eqn. (1), be at least be as good as static checks implementable with staging.
– In principle, one can embed a proof system $\vdash$ in the language by encoding proof trees as objects, and rules as functions, such that a proof object is constructible in the language only if the corresponding $\vdash$-proof is. Such proofs can be checked by the optimizer; and deductions made by the optimizer (such as $x + y = y + x$) can be used as premises for proofs. This is similar in spirit to the Curry-Howard isomorphism (e.g. [59]), although we embed proofs in values rather than types; and to proof-carrying code [60], although our approach is to intermingle proofs with the source code rather than having them separate.

– Such proof embeddings would let domain-specific libraries require *proof obligations* of users when automated static checks failed. For example, the expression $\mathsf{check}(x = y \ \lor \ P.\mathsf{proves}(\mathsf{equal}(x, y)))$ would succeed only if $x = y$ were proven by the optimizer, or if a proof object P were provided that gave a checkable proof of $x = y$.
– A reasonable test of a domain-specific safety checking system is whether it is powerful enough to subsume the type system. That is, one regards type checking as simply another kind of safety check to be implemented by a "type system library," an approach we've previously explored in [61]. Embedded type systems are attractive because they open a natural route to user-extensible type systems, and hence domain-specific type systems.

4 A Summing Up

We have argued that making programming languages more general-purpose is a useful research direction for domain-specific programming environments. In our view, truly general-purpose languages should let libraries provide domain-specific syntax extensions, performance improvements, and safety checks. Guaranteed optimization is an intriguing technology because it reduces abstraction penalty, can subsume staging, and its connections to theorem proving make it a good candidate for providing domain-specific safety checks.

The relationship between guaranteed optimization and other technologies providing staging-like abilities is summarized by this table:

	Annotation-free	Guarantees	Theorem Proving
Staging	○	●	○
Partial Evaluation	●	○	○
General Partial Computation	●	○	●
Guaranteed Optimization	◐	●	●

Staged languages (e.g., [21, 22]) are explicitly annotated with binding times, and have the advantage of guaranteeing evaluation of static computations at compile-time. The binding-time annotation is generally a *congruent division*, which effectively prevents theorem-proving about dynamic values.

Partial evaluation (e.g., [23]) automatically evaluates parts of a program at compile-time; in this respect it is closely related to guaranteed optimization. Partial evaluation is not usually understood to include proven guarantees of what will be statically evaluated; indeed, a lot of interesting research looks at effective heuristics for deciding what to evaluate. General partial computation [62] is an intriguing extension to partial evaluation in which theorem proving is used to reason about dynamic values.

Guaranteed optimization is largely annotation-free, although one must introduce some small annotations to control unfolding in the compile-time stage. It provides proven guarantees of what optimizations it will perform, and has the ability to prove theorems about run-time values.

4.1 Acknowledgments

We thank Andrew Lumsdaine and the referees for helpful discussions and comments on drafts of this paper.

References

1. Fischer, B., Visser, E.: Retrofitting the autobayes program synthesis system with concrete syntax (2004) In this volume.
2. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys **28** (1996) 196–196
3. Thomas, W.: Logic for computer science: The engineering challenge. Lecture Notes in Computer Science **2000** (2001) 257–267
4. Stepanov, A.: Abstraction penalty benchmark (1994)
5. Robison, A.D.: The abstraction penalty for small objects in C++. In: POOMA'96: The Parallel Object-Oriented Methods and Applications Conference. (1996) Santa Fe, New Mexico.
6. Müller, M.: Abstraction benchmarks and performance of C++ applications. In: Proceedings of the Fourth International Conference on Supercomputing in Nuclear Applications. (2000)
7. Wolfe, M.J.: High Performance Compilers for Parallel Computing. Addison Wesley, Reading, Mass. (1996)
8. Robison, A.D.: Impact of economics on compiler optimization. In: ISCOPE Conference on ACM 2001 Java Grande, ACM Press (2001) 1–10
9. Veldhuizen, T.L., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In: Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98), SIAM Press (1999)
10. Czarnecki, K., Eisenecker, U., Glück, R., Vandevoorde, D., Veldhuizen, T.: Generative programming and active libraries (extended abstract). In Jazayeri, M., Musser, D., Loos, R., eds.: Generic Programming '98. Proceedings. Volume 1766 of Lecture Notes in Computer Science., Springer-Verlag (2000) 25–39
11. Botorog, G.H., Kuchen, H.: Efficient parallel programming with algorithmic skeletons. In L. Bouge, P. Fraigniaud, A. Mignotte, Y.R., ed.: Proceedings of EuroPar '96. Volume 1123 of LNCS., Springer (1996) 718–731
12. Küchen, H.: Optimizing sequences of skeleton calls (2004) In this volume.
13. Lamping, J., Kiczales, G., Rodriguez, L., Ruf, E.: An architecture for an open compiler. In Yonezawa, A., Smith, B.C., eds.: Proceedings of the International Workshop on New Models for Software Architecture '92: "Reflection and Meta-level Architecture". (1992)
14. Chiba, S.: A Metaobject Protocol for C++. In: OOPSLA'95. (1995) 285–299
15. Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H., Konaka, H., Maeda, M., Kubota, K.: Design and implementation of metalevel architecture in C++ – MPC++ approach. In: Reflection'96. (1996)
16. Engler, D.R.: Incorporating application semantics and control into compilation. In: USENIX Conference on Domain-Specific Languages (DSL'97). (October 15–17, 1997)
17. Guyer, S.Z., Lin, C.: An annotation language for optimizing software libraries. In: Domain-Specific Languages. (1999) 39–52

18. Visser, E., Benaissa, Z.e.A., Tolmach, A.: Building program optimizers with rewriting strategies. ACM SIGPLAN Notices **34** (1999) 13–26 Proceedings of the International Conference on Functional Programming (ICFP'98).
19. Elliott, C., Finne, S., de Moore, O.: Compiling embedded languages. In Taha, W., ed.: Semantics, Applications, and Implementation of Program Generation. Volume 1924 of Lecture Notes in Computer Science., Montreal, Springer-Verlag (2000) 9–27
20. Schupp, S., Gregor, D., Musser, D., Liu, S.M.: User-extensible simplification–type-based optimizer generators. In: Proceedings of Compiler Construction 2001. Volume 2027. (2001) 86–101
21. Nielson, F., Neilson, H.R.: Two-Level Functional Languages. Cambridge University Press, Cambridge, Mass. (1992)
22. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical Computer Science **248** (2000) 211–242
23. Jones, N.D., Nielson, F.: Abstract interpretation. In Abramsky, S., Gabbay, D., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science. Volume 4. Oxford University Press (1995) To appear.
24. Kelsey, R., Clinger, W., (Editors), J.R.: Revised^5 report on the algorithmic language Scheme. ACM SIGPLAN Notices **33** (1998) 26–76
25. Meyer, B.: Eiffel: The Language. Prentice Hall (1991)
26. Evans, D., Guttag, J., Horning, J., Tan, Y.: LCLint: a Tool for Using Specifications to Check Code. In Wile, D., ed.: Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering. Volume 19:5 of ACM SIGSOFT Software Engineering Notes., New Orleans, USA (1994) 87–96
27. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of Operating Systems Design and Implementation (OSDI). (2000)
28. Leino, K.R.M.: Extended static checking: A ten-year perspective. Lecture Notes in Computer Science **2000** (2001) 157–175
29. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press (2002) 1–3
30. Owre, S., Rushby, J., Shankar, N., Stringer-Calvert, D.: PVS: an experience report. In Hutter, D., Stephan, W., Traverso, P., Ullman, M., eds.: Applied Formal Methods—FM-Trends 98. Volume 1641 of Lecture Notes in Computer Science., Boppard, Germany, Springer-Verlag (1998) 338–345
31. Spivey, J.M.: The Z Notation: A Reference Manual. Second edn. International Series in Computer Sciences. Prentice-Hall, London (1992)
32. Jones, C.B.: Systematic Software Development Using VDM. Second edn. Prentice-Hall International, Englewood Cliffs, New Jersey (1990) ISBN 0-13-880733-7.
33. Flanagan, C., Abadi, M.: Types for safe locking. In: European Symposium on Programming. (1999) 91–108
34. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, ACM Press (2002) 211–230
35. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: SIGPLAN Conference on Programming Language Design and Implementation. (1998) 249–257
36. Volpano, D.M., Smith, G.: A type-based approach to program security. In: TAPSOFT. (1997) 607–621

37. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, ACM Press (2002) 234–245
38. Kennedy, A.: Dimension types. In Sannella, D., ed.: Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming. Volume 788., Edinburgh, U.K., Springer (1994) 348–362
39. Veldhuizen, T.L.: C++ templates as partial evaluation. In Danvy, O., ed.: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. Technical report BRICS-NS-99-1, University of Aarhus, San Antonio, Texas, University of Aarhus, Dept. of Computer Science (1999) 13–18
40. Karmesin, S., Crotinger, J., Cummings, J., Haney, S., Humphrey, W., Reynders, J., Smith, S., Williams, T.: Array design and expression evaluation in POOMA II. In: ISCOPE'98. Volume 1505., Springer-Verlag (1998) Lecture Notes in Computer Science.
41. Veldhuizen, T.L.: Arrays in Blitz++. In: Computing in Object-Oriented Parallel Environments: Second International Symposium (ISCOPE 98). Volume 1505 of Lecture Notes in Computer Science., Springer-Verlag (1998) 223–230
42. Siek, J.G., Lumsdaine, A.: The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In: International Symposium on Computing in Object-Oriented Parallel Environments. (1998)
43. Neubert, T.: Anwendung von generativen programmiertechniken am beispiel der matrixalgebra. Master's thesis, Technische Universität Chemnitz (1998) Diplomarbeit.
44. McNamara, B., Smaragdakis, Y.: Static interfaces in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)
45. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)
46. Brown, W.E.: Applied template metaprogramming in SIUnits: The library of united-based computation. In: Second Workshop on C++ Template Programming. (2001)
47. Horn, K.S.V.: Compile-time assertions in C++. C/C++ Users Journal (1997)
48. Järvi, J., Willcock, J., Hinnant, H., Lumsdaine, A.: Function overloading based on arbitrary properties of types. C/C++ Users Journal **21** (2003) 25–32
49. Veldhuizen, T.L.: Active Libraries and Universal Languages. PhD thesis, Indiana University Computer Science (2004) (forthcoming).
50. Veldhuizen, T.L., Lumsdaine, A.: Guaranteed optimization: Proving nullspace properties of compilers. In: Proceedings of the 2002 Static Analysis Symposium (SAS'02). Volume 2477 of Lecture Notes in Computer Science., Springer-Verlag (2002) 263–277
51. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems **13** (1991) 181–210
52. Click, C., Cooper, K.D.: Combining analyses, combining optimizations. ACM Transactions on Programming Languages and Systems **17** (1995) 181–196
53. Wei, J.: Correctness of fixpoint transformations. Theoretical Computer Science **129** (1994) 123–142
54. Courcelle, B., Kahn, G., Vuillemin, J.: Algorithmes d'équivalence et de réduction à des expressions minimales dans une classe d'équations récursives simples. In Loeckx, J., ed.: Automata, Languages and Programming. Volume 14 of Lecture Notes in Computer Science., Springer Verlag (1974) 200–213

55. Milner, R.: Communication and Concurrency. International Series in Computer Science. Prentice Hall (1989)
56. Rutten, J.J.M.M.: Universal coalgebra: a theory of systems. Theoretical Computer Science **249** (2000) 3–80
57. Veldhuizen, T.L., Siek, J.G.: Combining optimizations, combining theories. Technical Report TR582, Indiana University Computer Science (2003)
58. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems (TOPLAS) **1** (1979) 245–257
59. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard isomorphism. Technical report TOPPS D-368, Univ. of Copenhagen (1998)
60. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, France (1997)
61. Veldhuizen, T.L.: Five compilation models for C++ templates. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)
62. Futamura, Y., Nogi, K.: Generalized partial computation. In Bjørner, D., Ershov, A.P., Jones, N.D., eds.: Proceedings of the IFIP Workshop on Partial Evaluation and Mixed Computation, North-Holland (1987)