Interlanguage Migration: From Scripts to Programs

Sam Tobin-Hochstadt
Northeastern University
Boston, MA
samth@ccs.neu.edu

Matthias Felleisen
Northeastern University
Boston, MA
matthias@ccs.neu.edu

ABSTRACT

As scripts grow into full-fledged applications, programmers should want to port portions of their programs from scripting languages to languages with sound and rich type systems. This form of interlanguage migration ensures type-safety and provides minimal guarantees for reuse in other applications, too.

In this paper, we present a framework for expressing this form of interlanguage migration. Given a program that consists of modules in the untyped lambda calculus, we prove that rewriting one of them in a simply typed lambda calculus produces an equivalent program and adds the expected amount of type safety, i.e., *code in typed modules can't go wrong*. To ensure these guarantees, the migration process infers constraints from the statically typed module and imposes them on the dynamically typed modules in the form of behavioral contracts.

1. WHEN SCRIPTS GROW UP

In the beginning, the programmer created a script to mechanize some routine but problematic task. The script consisted of a few dozen lines of code in a dynamically typed and expressive scripting language. Before long, the programmer discovered that friends were coping with similar problems, and with a few changes here and a few hacks there, the script became a useful 1,000-line program for his friends, too. Not surprisingly, the programmer decided to offer this service to some of his favorite mailing lists and modified the program a few more times. At that point, there were actually several related applications; each consisted of a few dozen components, mostly drawn from a common library of modules; and all these applications supported different tasks, only superficially related to the original one. It goes without saying that the application suite became the core of a small, yet exponentially growing business and that the programmer sold it for a lot of money to some big company.

Fortunately, our programmer has employed a test-driven or test-first approach to programming [5], a technique that originated with dynamically typed languages such as Lisp, Scheme and Smalltalk. [1] Every bug report has been turned into a test case; the test suites have been maintained in a meticulous manner. Unfortunately the big company isn't satisfied with the result. Because the software deals with people's financial holdings, the company's management team wants to cross all the t's and dot all the i's, at least for the critical modules. The team has set the goal to rewrite the program in a programming language with a sound type system. They believe that this step will eliminate some long-standing bugs from the modules and improve the efficiency of the debugging team. After all, "typed programs can't go wrong" [24], i.e., the programmer doesn't have to look at code in typed modules when a run-time type check fails.

Currently, this common, realistic situation poses a major difficulty for the (unfortunate) programmers of the company. Ideally, they should port one module at a time, always leaving the overall product intact and running. Most foreign-language interfaces, however, support only connections between high-level languages and C-level libraries. Support for connecting a high-level typed language with a high-level dynamically typed language rarely exists. Hence, programmers often re-develop the entire program from scratch and run into Brooks's "second system" syndrome [8].

In this paper, we investigate an alternative to this reimplementation approach. Specifically, we present a framework for porting programs in a gradual manner from an dynamically typed to a syntactically and semantically related, statically typed programming language. We are making three philosophical assumptions. First, a program is a sequence of modules in a safe, but dynamically typed programming language. The second assumption is that we have an explicitly, statically typed programming language that is variant of the dynamically typed language. Specifically, the two languages share run-time values and differ only in that one has a type system and the other doesn't. While there have been few realistic examples of such systems [7], one can imagine creating such a pair of languages from any dynamically typed language (say Scheme), possibly based on the decade-old soft typing research. Finally, a third and less realistic assumption is that (at least) one of the modules is "typable," meaning that equipping variables with type dec-

[1]Although Kent Beck has coined the phrase and popularized test-driven development, the idea of developing tests first had been around for ages in the Scheme community and had been inherited from the Lisp community. For example, see the unpublished manuscript "A Guide to the Metaphysical Universe" by Friedman, Haynes, Kohlbecker, and Wand ca. 1984.

larations suffices to port the module from the untyped to the typed version of the language. We realize that "typing" a module may require a serious rewriting effort, possibly supported by tools, but for our purposes, this simplifying assumption suffices.

Our primary goal is to establish a Milner-style theorem for the process of adding types to a program:

> typed modules can't go wrong, and all run-time errors originate in untyped modules.

While we initially thought that such a theorem would "obviously" hold, we soon realized that even our simple model turned out to pose research questions. A modicum of reflection suggests that a soundness guarantee (such as the above) demands some protection of the typed module(s) from the dynamically typed ones. First, a dynamically typed module may abuse a typed function imported from a typed module. Second, the typed module may import functions from untyped modules and those may violate the type discipline of the module. Finally, because of higher-order functions, which are found in most popular scripting languages these days, both the first and the second problem aren't separable and most be solved together.

For a first impression, consider a concrete example:

```
(module f  ;; declare and export f
        (int -> int) ;; with type int to int
        ;; as follows:
        (lambda (x int)
          (g x)))
```

If this typed module is to export a value of type (**int** → **int**), `g` must be a function that maps integers to integers. Unfortunately, the module from which `f` imports `g` provides no such guarantees. We solve this problem with a type checker that infers constraints from the typed module on the remainder of the modules. Typically such constraints are written down in the documentation of a module but are not checked. Once we have derived the constraints, we turn them into *behavioral contracts* [11] for the dynamically typed modules. Since that is still insufficient for safety, we also generate wrapper modules that import functions from dynamically typed modules with contracts and re-export them with more stringent contracts. To keep things manageable for programmers, we avoid this second step as much as possible, since it introduces unexpected new modules into the code base.

The paper consists of six sections. The second section gives a high-level overview of interlanguage migration, plus the supporting contract and module systems. A formal model of the interlanguage migration is introduced in the third section and proved sound in the fourth. In the fifth section we discuss related work.

2. AN INFORMAL TOUR

A typical scripting language is a highly expressive but untyped programming language. Usually scripting languages are (intended to be) safe. Almost all the popular ones include a module-like facility. To make things simple, we assume that a program is a sequence of modules followed by a "main expression."

The evaluation of such a program starts with the main expression. Evaluation proceeds as usual in an expression-oriented language. When the evaluation process encounters a reference to a module, it imports the appropriate expression from there and evaluates it.

To keep things as simple as possible, we work with a typed variant of the untyped language where all binding positions in the come with type declarations. In short, we migrate to an explicitly typed language that is otherwise syntactically and semantically equivalent to the untyped one.

In a program that mixes typed and untyped modules, evaluation proceeds as before. This implies that the values of the typed language are those of the untyped language (and vice versa). For the reader with formal inclinations, figures 1, 2, 3 and 4 present the formal syntax and semantics of our model, though in this section we liberally add features to illustrate our points; section 3 explains these figures in detail.

Given an untyped modular program, the first step of interlanguage migration is to turn one module into a typed module. We assume that this step simply adds types to all binding positions of the module, including the module exports. Naturally, we don't believe that this is possible in reality. Instead, we expect that programmers rewrite their modules in a meaning-preserving manner—possibly using software analysis tools—so that they can add type declarations where needed. Still, because the goal of migration is to change the program without changing its meaning and our goal is to understand the effect of the overall process, we consider it justified for a first framework to simplify this step as much as possible.

After the chosen module has been rewritten in the typed version of the language, we need to check the types and infer from them how the typed module is going to interact with the others, which remain untyped. Consider the following simplistic program:

```
(module f (int -> int) (lambda (x int) (g x)))
(module g 999)
(f 5)
```

It consists of two modules: the first is presumably a module that has been rewritten in the typed language, the second one is still in the untyped language. Also, the first one exports a function from integers to integers; the second one exports a simple integer.

If we were to evaluate this program as is, it would eventually attempt to apply `999` to `5` via the application $(g\ x)$ in the typed module. In other words, the typed portion of the program would trigger a run-time error, which, assuming proper source tracking, would tell the programmer that the typed module went wrong.

A different view of the problem is that when one module changes, the rest of the program has to play by new rules, too. In this case, the very fact that the export from `g`, the second module, is used as a function in the typed module establishes an agreement between the two modules. This agreement, however, is informal (and unuttered) and is not monitored during run-time. The evaluation therefore results in a run-time error seemingly due to the typed module.

Thus our first lesson is that informal agreements don't jive with the goal of introducing types. To reap the benefits of types, we must not only have agreements, we must enforce them. This line of reasoning naturally suggests the use of behavioral contracts in the spirit of Findler and Felleisen [11]. More precisely, we assume that an interlanguage migration process has access to the interfaces of the remaining modules and that it is possible to add contracts to these interfaces.

$P ::= e \mid MP$	Programs
$M ::= (\textbf{module}\ f\ v)$	Modules
$v ::= n \mid (\lambda x.e)$	Values
$e ::= v \mid x \mid f \mid (e\ e) \mid (\textbf{if0}\ e\ e\ e)$	Expressions

Figure 1: Scripting Language Syntax

$P ::= e_m \mid MP$	Programs
$M ::= M_c \mid M_u \mid M_t$	Modules
$M_u ::= (\textbf{module}\ f\ v)$	Untyped Modules
$M_c ::= (\textbf{module}\ f\ c\ v)$	Contracted Modules
$M_t ::= (\textbf{module}\ f\ t\ v_t)$	Typed Modules
$c ::= \textbf{int} \mid (c \rightarrow c) \mid \textbf{int} \vee (c \rightarrow c)$	Contracts
$t ::= \textbf{int} \mid (t \rightarrow t)$	Types $\subseteq$ Contracts
$v ::= n \mid (\lambda x.e)$	Untyped Values
$v_t ::= n \mid (\lambda x : t.e_t)$	Typed Values
$v_m ::= n \mid (\lambda x : t.e_m) \mid (\lambda x.e_m)$	Mixed Values
$e ::= v \mid x \mid f \mid (e\ e) \mid (\textbf{if0}\ e\ e\ e)$	Untyped Expressions
$e_t ::= v_t \mid x \mid f \mid (e_t\ e_t) \mid (\textbf{if0}\ e_t\ e_t\ e_t)$	Typed Expressions
$e_m ::= v_m \mid x \mid f \mid (e_m\ e_m) \mid (\textbf{if0}\ e_m\ e_m\ e_m)$	Mixed Expressions

Figure 2: Typed Language Syntax

For our running example we would expect that migration changes the program as follows:

```
(module f (int -> int) (lambda (x int) (g x)))
(module g (integer? -> integer?) 999)
(f 5)
```

Put differently, we can infer from the types of the first module that the second module must always export a function from integers to integers. In our framework, we express this fact with a contract to the module interface.

The example has two implications for interlanguage migration and its formal model. First, the language must also include optional contracts at module boundaries. Second, a type checker for the typed variant of the language must not only enforce the rules of the type system, it must also infer the contracts that these type annotations imply for the remaining modules.

Unfortunately, there are yet more problems. Consider this second program:

```
(module h (int -> int)
  (lambda (y int)
    (let ((g (lambda (x int) (+ x 10))))
      (+ (g y) (g 10)))))
(h false)
```

Here the programmer applies an (**int** → **int**) function to `false`, a boolean. Since our evaluator ignores types, the boolean value flows into the typed module without any objections, only to cause havoc there. Again, the typed module appears to have gone wrong.

In this case, the solution is to interpret the types on the module exports as contracts so that the evaluator monitors how the other modules use functional exports from the typed module. For flat types such as **int**, the values that flow into typed functions are checked immediately; for functional values, the contracts are distributed over the domain and range of the function until flat values show up [11]. Technically, the types become contracts on external references to the module `f`, and are interpreted as runtime checks, or casts, which specify the party to be blamed if they fail:

```
(h false)
```

steps to

```
({(integer? -> integer?) <= h : Main} false)
```

steps to

```
{integer? <= (h {integer? <= false : Main}) : Main}
```

At this point it has become clear that `false` is a bad value, and the evaluator can abort the execution blaming the main expression for supplying bad values to the typed module.

Simply adding contracts to existing modules doesn't solve all problems, though:

```
(module f int
  (if (not (m 5))
      (m true)
      7))
(module m ((or/c boolean? integer?)
            ->
           (or/c boolean? integer?))
  (lambda (x)
    (if (boolean? x) true (+ x 1))))
f
```

The first, typed module contains two references to `m`, the second, untyped module. From the types of the subexpressions we can infer that `m` must export a function that can consume both booleans and integers and that can also produce both kinds of values. The resulting contract uses `or/c` in the domain and range part to state this fact but it still means that the evaluation of `f` raises a run-time error because `(m true)` produces a boolean rather than a number.

Our solution is to add new wrapper modules with more specific contracts based on the existing type and contract information:

$$
\begin{array}{ll}
v ::= n \;\mid\; (\lambda x : t.e_t) \;\mid\; (\lambda x.e) \;\mid\; \{(c \dashrightarrow c) \Leftarrow^{f} v\} & \text{Values} \\
e ::= v \;\mid\; x \;\mid\; f \;\mid\; (e\ e) \;\mid\; (\mathbf{if0}\ e\ e\ e) \;\mid\; \{c \Leftarrow^{f} e_t\} & \text{Expressions}
\end{array}
$$

Figure 3: Runtime Syntax

$$
\begin{array}{lcll}
((\lambda x.e)\ v) & \longrightarrow & [v/x]e & \textsc{Subst} \\
((\lambda x : t.e)\ v) & \longrightarrow & [v/x]e & \textsc{TypedSubst} \\
(n\ v)^{f} & \longrightarrow & (\mathbf{blame}\ f) & \textsc{App-Error} \\
(\mathbf{if0}\ 0\ e_1\ e_2) & \longrightarrow & e_1 & \textsc{if0-True} \\
(\mathbf{if0}\ v\ e_1\ e_2) & \longrightarrow & e_2 & \textsc{if0-False} \\
\{\mathbf{int} \Leftarrow^{g} n\} & \longrightarrow & n & \textsc{Int-Int} \\
\{\mathbf{int} \vee c \Leftarrow^{g} n\} & \longrightarrow & n & \textsc{Int-IntOr} \\
\{\mathbf{int} \vee c \Leftarrow^{g} v\} & \longrightarrow & \{c \Leftarrow^{g} v\} & \textsc{Int-LamOr} \\
\{\mathbf{int} \Leftarrow^{g} v\} & \longrightarrow & (\mathbf{blame}\ g) & \textsc{Int-Lam} \\
\{(c_1 \rightarrow c_2) \Leftarrow^{g} n\} & \longrightarrow & (\mathbf{blame}\ g) & \textsc{Lam-Int} \\
\{(c_1 \rightarrow c_2) \Leftarrow^{g} v\} & \longrightarrow & \{(c_1 \dashrightarrow c_2) \Leftarrow^{g} v\} & \textsc{Lam-Lam} \\
(\{(c_1 \dashrightarrow c_2) \Leftarrow^{g} v\}\ w)^{f} & \longrightarrow & \{c_2 \Leftarrow^{g} (v\ \{c_1 \Leftarrow^{f} w\})\} & \textsc{Split} \\
\ldots(\mathbf{module}\ f\ v)\ldots E[f] & \longrightarrow & \ldots(\mathbf{module}\ f\ v)\ldots E[v] & \textsc{Lookup} \\
\ldots(\mathbf{module}\ f\ c\ v)\ldots E[f] & \longrightarrow & \ldots(\mathbf{module}\ f\ c\ v)\ldots E[\{c \Leftarrow^{f} v\}] & \textsc{Lookup-Contract} \\
\ldots(\mathbf{module}\ f\ t\ v)\ldots E[f^{g}] \text{ where } g \neq f & \longrightarrow & \ldots(\mathbf{module}\ f\ t\ v)\ldots E[\{t \Leftarrow^{g} v\}] & \textsc{Lookup-Type} \\
\ldots(\mathbf{module}\ f\ t\ v)\ldots E[f^{f}] & \longrightarrow & \ldots(\mathbf{module}\ f\ t\ v)\ldots E[v] & \textsc{Lookup-Type-Self}
\end{array}
$$

Figure 4: Reduction Rules

```
(module f int
  (if (not (m-bool 5))
      (m-int true)
      7))

(module m-int
     ((or/c boolean? integer?) -> integer?)
     m)

(module m-bool
     ((or/c boolean? integer?) -> boolean?)
     m)

(module m ((or/c boolean? integer?)
           ->
           (or/c boolean? integer?))
  (lambda (x)
    (if (boolean? x) true (+ x 1))))

f
```

The new wrapper modules `m-int` and `m-bool` provide additional guarantees about the behavior of `m`, by placing stricter runtime contracts on the body of `m` than the contracts that were originally inferred. The new contracts allow the original module `f` to typecheck. When the program is evaluated, the contracts show that `m-bool` misrepresented `m`, and is thus blamed for the runtime violation.

This strategy raises the question: why we do not add wrapper modules everywhere, that contracts add safety checks for particular variable occurrences? This solution, while conceptually simple, fails to generate maintainable code. Each time this migration step is applied (for example, for each module ported from being untyped to typed), a slew of new wrapper modules would be created. Soon, the system would be an incomprehensible mess.

This points to a design requirement we consider fundamental: that the resulting program be maintainable. Our transformation is not a compilation strategy to ensure safety, it is a migration that is part of the development of a system. As with a semantics-preserving refactoring, it must respect the surrounding code as much as possible.

The rest of the paper presents a formal model of this migration process, drawing on our experience of implementing the model as a prototype. The focus of our presentation concerns the derivation of constraints via type checking; the translation of these constraints into contracts; the addition of wrapper modules based on types and contracts; and last but not least a theorem that proves that typed modules can't go wrong in this setting.

3. THE FORMAL FRAMEWORK

The objective of this section is to describe our interlanguage migration framework formally, from the syntax and semantics of the programming languages all the way to the linguistic aspects of the process itself.

3.1 Syntax

Our scripting language is a simplified version of the language of Meunier *et al* [23], augmented with types and typed modules. It consists of the lambda calculus enriched with numeric constants and a conditional, as well as casts, modules and contracts. The initial syntax used in the original program is specified in figure 1. After the migration step, the syntax is more complicated: see figure 2. The runtime system collapses some distinctions and adds casts, which are specified in figure 3.

MODULES Our language has a simple first-order module system, in which each module consists of a name and a value. The module exports its value via its name. Two other module forms are provided: modules with contracts and modules with types. Contracted modules are identical to their untyped counterparts, except that a contract is added to the (simplified) module interface. When the value of the module is used, that value is checked against the contract. Typed modules have a top-level type, and contain only typed values, v_t in figure 2.

Contracts and Casts The contracts allow the base **int** contract, as well as function contracts and disjunction. Function contracts have the Findler–Felleisen semantics [11], and disjunction allows either of the two branches to be satisfied. The disjunction of two function contracts is syntactically prohibited. This restriction significantly reduces the complexity of the reduction rules for $\vee$-contracts. At run-time, contracts turn into checks, which we express with casts. Syntactically, a cast $\{c \Leftarrow^{e} m\}$ combines a contract with an expression and a label for a module, which it blames for the contract violation if the check fails.

Casts are not part of the source language, but the state space of the runtime system; programmers cannot write them and our transformation does not insert them.

Types The types of the typed fragment of the language are just the base type **int** and function types. Importantly, every type is syntactically also a contract.

Expressions The language contains three kinds of expressions: typed, untyped and mixed. Typed expressions occur only in the typed module. Untyped expressions occur in all other modules. Mixed expressions can contain typed and untyped subexpressions and appear *only* in the main expression. The main expression is initially untyped.

3.2 Semantics

For the dynamic semantics, we assume that every expression has been labeled with the name of its original source module. The main expression is labeled with m. This labeling is necessary for appropriate blame assignment when a dynamic error occurs. It corresponds to an annotation pass for source location tracking. For clarity, we omit these labels wherever they are not needed.

The dynamic semantics is defined in figure 4 as a reduction semantics, and again follows Meunier *et al*, with additions for types and $\vee$-contracts. Reduction takes place in the context of modules, which are not altered during reduction. The relation $\rightarrow$ is the one-step reduction relation, with $\rightarrow^*$ as its reflexive, transitive closure, and the set of evaluation contexts is defined as:

$$E = [] \mid (E\ e) \mid (v\ E) \mid (\mathbf{if0}\ E\ e\ e) \mid \{c \Leftarrow^{f} E\}$$

Rules that do not refer explicitly to the context are implicitly wrapped in $E[-]$ on both sides, with the exception of the rules that reduce to $(\mathbf{blame}\ f)$, which discard the evaluation context.

The reduction rules fall into the following categories:

- The rules that lookup module references all refer explicitly to the module context. The Lookup rule refers to untyped modules, and simply substitutes the body of the module for the reference. The LookupContract and LookupType rules retrieve the appropriate expression and wrap it in a contract. The contract wrapped around typed module bodies is necessary so that typed expressions are never used in incorrect ways, even when the untyped modules refer to the typed module. This check is not necessary when the typed module refers to itself, and is thus omitted in the LookupTypeSelf rule [11].

- The rules for the λ-calculus core are straightforward. These include Subst and TypedSubst, which perform β_v-reduction on untyped and typed abstraction respectively. IF0-True and IF0-False are also simple.

- App-Error is the one runtime error that does not involve a contract or a cast. If a number is in the application position of an application, clearly the invariants of the language have been violated. We blame the source of the application for this error.

  The remaining rules handle contracts and casts.

- Int-Int and Int-IntOr pass numbers through **int** contracts unchanged. Int-LamOr discards an unsatisfiable disjunct. We do not need rules for disjunction with an arrow contract on the left, since such contracts are syntactically prohibited.

- Int-Lam and Lam-Int represent contract failures and blame the appropriate party, as labeled on the cast.

- Lam-Lam blesses an arrow contract applied to an abstraction, turning it into a blessed arrow contract. In contrast to Int-Int, we must keep the contract around for later use. The resulting expression, while still a cast, is also a syntactic value.

- The Split rule breaks a blessed arrow contract into its positive and negative halves, and places them around the argument and the entire application. The creation of two new casts, with appropriate blame assignment, is the key to proper contract checking for higher-order functions [11].

3.3 Adding Type Declarations

The first step in interlanguage migration requires the programmer to change one module from the untyped language to the typed one. In our system, this involves adding types to every variable binding and to the module export as a whole.

Once this module is annotated, the new program is referred to as P^{M}, where M is the name of the now-typed module.

Since the simply-typed λ-calculus has a straightforward type-soundness theorem, we might expect a similar one to hold for migrated programs, provided the type annotations are self-consistent. Sadly, this is not the case. For example, the typed module M might refer to some other module, which could provide an arbitrary value or raise a runtime error. The other modules may contain outright errors, such as $(3\ 4)$, as well as untypeable expressions such as $(\lambda x.(\mathbf{if0}\ x\ 1\ (\lambda y.y)))$; we do not rule these out, since the programmer is only adding types to one module. Furthermore, since we do not typecheck the other modules, they may use the typed module in ways that do not accord with its type. Because of these possibilities, the migration process must protect the typed module from its untyped brethren.

$$\frac{}{\Gamma \vdash^{RT} x : \Gamma(x); \emptyset}\ \textsc{MT-Var} \qquad \frac{}{\Gamma \vdash^{RT} f : t; \{f \lhd t\}}\ f \neq M\ \textsc{MT-ModVar} \qquad \frac{}{\Gamma \vdash^{RT} M : t; \emptyset}\ \text{if module } M \text{ has type } t.\ \textsc{MT-ModVarSelf}$$

$$\frac{}{\Gamma \vdash^{RT} n : \mathbf{int}; \emptyset}\ \textsc{MT-Int} \qquad \frac{\Gamma, x : t \vdash^{RT} e : s; \phi}{\Gamma \vdash^{RT} (\lambda x : t.e) : (t \rightarrow s); \phi}\ \textsc{MT-Abs}$$

$$\frac{\Gamma \vdash^{RT} e_1 : (t_1 \rightarrow t_2); \phi_1 \qquad \Gamma \vdash^{RT} e_2 : t_1; \phi_2}{\Gamma \vdash^{RT} (e_1\ e_2) : t_2; \phi_1 \cup \phi_2}\ \textsc{MT-App} \qquad \frac{\Gamma \vdash^{RT} e_1 : t_1; \phi_1 \qquad \Gamma \vdash^{RT} e_2 : t_2; \phi_2 \qquad \Gamma \vdash^{RT} e_3 : t_2; \phi_3}{\Gamma \vdash^{RT} (\mathbf{if0}\ e_1\ e_2\ e_3) : t_2; \phi_1 \cup \phi_2 \cup \phi_3}\ \textsc{MT-If0}$$

Figure 5: Constraint Generation

3.4 Inferring Constraints from Types

In order to protect the typed module when it refers to untyped ones, we apply a transformation to the program that expresses the implicit agreements between the typed and untyped modules via contracts. Our transformation examines the references to other modules in M and from these uses infers contracts that become obligations of those other modules. For example, in the following program, the context makes it obvious that g must have type $(\mathbf{int} \rightarrow \mathbf{int})$:

```
(module f int (g 5))
```

The transformation would therefore add the contract $(\mathbf{int} \rightarrow \mathbf{int})$ to g.

The type system in figure 5 formalizes this intuition. Its rules define the judgement

$$\Gamma \vdash^{RT} e : t; \Phi$$

which states that in type environment Γ, expression e has type t under the constraint set Φ. The rules are similar to those of the simply typed λ-calculus, propagating and combining the constraints from their constituents, except for the module variable reference rules: MT-ModVarSelf and MT-ModVar. The former checks references to the typed module itself. The latter allows a module variable to be assigned any type and adds a matching constraint to the constraint set. Constraints are of the form $g \lhd c$, which states that g must have contract c.

Because of the non-determinism of the MT-ModVar rule, these rules do not naturally map onto a syntax-directed type checker. They do define a logic program, however, which potentially produces many solutions, each satisfying the desired type, and including a matching set of constraints. Each of these solutions gives rise to a potentially different set of contracts imposed on the other modules in the program. For example, consider the following program:

```
(module f int (g h))
```

There are many possible sets of constraints that could be generated by our system. A simple one requires that h be an **int** and that g have the contract $(\mathbf{int} \rightarrow \mathbf{int})$. Of course, there are infinitely many possibilities. While a real system for migration would use programmer input and static analysis to choose one solution, our soundness theorem holds for all of them. We have prototyped this system in Schelog [27], an embedding of Prolog in Scheme. In our prototype, we simply choose the first solution produced. In real programs, we conjecture that the contracts are over- not under-determined, and that migration tools like these will need input from programmers to decide whether to reject a program with conflicting constraints, or to change some portion of the original program.

$$\begin{aligned}
merge(c, c) &= c \\
merge(\mathbf{int}, c) &= \mathbf{int} \vee c \\
merge(c, \mathbf{int}) &= \mathbf{int} \vee c \\
merge((c_1 \rightarrow c_2), (c_3 \rightarrow c_4)) &= \\
&(merge(c_1, c_3) \rightarrow merge(c_2, c_4))
\end{aligned}$$

Figure 6: From Constraints to Contracts

3.5 From Constraints to Contracts

The next step is to turn the set of constraints into an actual contract. For example, the constraint set for our first example is $\{g \lhd (\mathbf{int} \rightarrow \mathbf{int})\}$. The obvious contract is then $(\mathbf{int} \rightarrow \mathbf{int})$ for module g. When there are multiple references to a module variable within the typed module, however, there are necessarily multiple constraints on that module, which we must somehow combine into a single contract.

Two obvious approaches present themselves. Consider the following program:

```
(module f (int -> int)
        (lambda (x int) ((g 1) (g 0))))
(module g (lambda (x) (if0 x 1 (lambda (y) y))))
f
```

Given our evaluation rules for **if0**, this is a perfectly reasonable untyped program. Furthermore, the untypeable expression is in g, to which we are not adding types.

The constraints generated for g are

$$\{g \lhd (\mathbf{int} \rightarrow \mathbf{int}), g \lhd (\mathbf{int} \rightarrow (\mathbf{int} \rightarrow \mathbf{int}))\}$$

First, if the language of contracts supported conjunction, the merge operation could just compute the conjunction of all constraints. This would give

$$(\mathbf{int} \rightarrow ((\mathbf{int} \rightarrow \mathbf{int}) \wedge \mathbf{int}))$$

as the contract for g. But no value is both a function and an integer, so this contract cannot possibly be satisfied. Since we want to allow this program, conjunction is not the correct solution.

Second, the merge can choose the disjunction of the constraints. In our example, we would get

$$(\mathbf{int} \rightarrow (\mathbf{int} \vee (\mathbf{int} \rightarrow \mathbf{int})))$$

which is legal and acceptable

The process of combining constraints into contracts is specified in figure 6. This process ensures that contracts are tidy in arrows: there is only one arrow contract in any disjunction. This invariant is required by our contract syntax, and simplifies the reduction rules for contracts with disjunction.

3.6 Adding Wrapper Modules

Unfortunately, disjunction in contracts introduces new problems. The contract we assign to g is not sufficient to establish that $(g\ 1)$ produces an abstraction, which is required for the body of f to execute without type errors. Therefore, we must place some additional constraint on $(g\ 1)$. To implement these constraints, we add a wrapper module to the program, with a more precise contract for g. To accomplish this, we perform a second pass over the typed module, accumulating unsatisfied constraints and changing module references to point to the new wrapper modules. Then the transformation adds the required wrappers to the program.

These rules have three interesting cases:

- AC-ModVarContract applies for module variables where the contract is sufficient to ensure that the type is always satisfied.
- AC-ModVarWrap requires a new wrapper module and changes the module reference to the new wrapper module, whose name is formed from the original name and the type, when the contract is insufficient.
- AC-ModVarSelf handles self-reference to the typed module M.

The other rules merely recur structurally.

Once we have a collection of constraints C, it is trivial to construct the appropriate wrapper modules. For every element $(f, t) \in C$, simply add a new module of the form

```
(module f-t t f)
```

to the original program.

This transformation relies on a relationship between contracts and types. We write $c \Rightarrow t$ when contract c establishes the preconditions for type t. Then we insert casts precisely at those module references f in M when $c \not\Rightarrow t$, where c is the contract on f and t is the desired type. We formalize this relation with two new judgements

$$\vdash c \Rightarrow t$$

and

$$\vdash c \Leftarrow t$$

These judgements are defined in figure 8. The first judgement states that contract c is sufficient to establish the preconditions of type t. The second is the converse, namely that t establishes the preconditions for c. The second judgement is not used in the rest of the transformation; it is only needed for the definition of the first.

With this definition, we can now define a further transformation on the typed module, which adds wrappers at the places we have just described. This transformation is presented in figure 7, which defines the judgement

$$P, \Gamma \vdash^{\Rightarrow} e : t; e'; c$$

This states that Γ proves that e can be transformed to e', which then has type t with wrapper modules generated from constraints c.

3.7 Summary of the transformation

The transformation we have just described is MT, for migration transformation. $MT(P)$ transforms P as follows:

1. The programmer chooses one module M from P and adds types to this module, so that is is now a typed module according to the grammar and type system, producing M'.
2. Using the type system described in figure 5, generate constraints from module M'.
3. Merge these constraints according to figure 6 to produce contracts, which are added to the other modules.
4. Transform M' into M'' to accommodate weak contracts.
5. Add wrapper modules as required by the transformation of M' to M''.

This gives a new program P^M.

Since step 2 in this process is nondeterministic, $MT(P)$ produces a set of programs. We prove in the next section that every element P^M of this set implements P.

4. SOUNDNESS

Before we can prove that our migration transformation is sound, we must first define what soundness means for a partially typed system. It cannot mean absence of runtime errors, since not all modules are necessarily typed. All we can say instead is that the *typed* modules do not go wrong.

4.1 Soundness for Mixed Programs

Soundness for interlanguage migration is a relation between the program before and after migration. Intuitively, this relation states that the two programs agree when they both produce values and that the typed module never produces a type error at runtime. We say that a program that has been migrated is *partially typed*.

Definition 1 (Soundness). *$P \trianglerighteq P^M$, where P^M is partially typed with typed module M iff:*

1. *If $P \rightarrow^* v$ then $\exists v'$ where $P^M \rightarrow^* v'$ with $v \triangleright v'$ or $P^M \rightarrow^* (\textbf{blame}\ g)$ with $g \neq M$.*
2. *If $P \rightarrow^* (\textbf{blame}\ h)$ then $\exists g$ where $P^M \rightarrow^* (\textbf{blame}\ g)$ with $g \neq M$.*
3. *If P reduces forever, then P^M reduces forever or $\exists g$ where $P^M \rightarrow^* (\textbf{blame}\ g)$ with $g \neq M$.*

This definition relies on the *similarity* relation $v \triangleright v'$, which states that v' is the same as v, with the possible addition of types and casts. In figure 9, this relation is defined formally and extended to modules and to programs. For programs, $P \triangleright P^M$ states that P and P^M are syntactically identical, ignoring casts, contracts and types.

We say that a system for typed migration is *sound* if the migrated program is always in the $\trianglerighteq$ relation to the original program.

$$\frac{\text{AC-ModVarContract}\quad (\textbf{module}\ f\ c\ e) \in P \qquad \vdash c \Rightarrow t}{P, \Gamma \vdash^{\Rightarrow} f : t; f; \emptyset}
\qquad
\frac{\text{AC-ModVarWrap}\quad (\textbf{module}\ f\ c\ e) \in P \qquad \vdash c \not\Rightarrow t}{P, \Gamma \vdash^{\Rightarrow} f : t; f\text{-}t; \{(f, t)\}}
\qquad
\frac{\text{AC-ModVarSelf}\quad (\textbf{module}\ M\ t\ e) \in P}{P, \Gamma \vdash^{\Rightarrow} M : t; M; \emptyset}$$

$$\text{AC-Var}\quad P, \Gamma \vdash^{\Rightarrow} x : \Gamma(x); x; \emptyset
\qquad
\text{AC-Int}\quad P, \Gamma \vdash^{\Rightarrow} n : \textbf{int}; n; \emptyset
\qquad
\frac{\text{AC-App}\quad P, \Gamma \vdash^{\Rightarrow} e_1 : (t_1 \to t_2); e_1'; C_1 \qquad P, \Gamma \vdash^{\Rightarrow} e_2 : t_1; e_2'; C_2}{P, \Gamma \vdash^{\Rightarrow} (e_1\ e_2) : t_2; (e_1'\ e_2'); C_1 \cup C_2}$$

$$\frac{\text{AC-If0}\quad P, \Gamma \vdash^{\Rightarrow} e_1 : t_1; e_1'; C_1 \qquad P, \Gamma \vdash^{\Rightarrow} e_2 : t_2; e_2'; C_2 \qquad P, \Gamma \vdash^{\Rightarrow} e_3 : t_2; e_3'; C_3}{P, \Gamma \vdash^{\Rightarrow} (\textbf{if0}\ e_1\ e_2\ e_3) : t_2; (\textbf{if0}\ e_1'\ e_2'\ e_3'); C_1 \cup C_2 \cup C_3}
\qquad
\frac{\text{AC-Abs}\quad P, \Gamma, x : t \vdash^{\Rightarrow} e : s; e'; C}{P, \Gamma \vdash^{\Rightarrow} (\lambda x : t.e) : (t \to s); (\lambda x : t.e'); C}$$

Figure 7: Generating Wrapper Modules

$$\vdash \textbf{int} \Rightarrow \textbf{int}
\qquad
\frac{\vdash c_1 \Leftarrow t \qquad \vdash c_2 \Rightarrow s}{\vdash (c_1 \to c_2) \Rightarrow (t \to s)}$$

$$\vdash \textbf{int} \Leftarrow \textbf{int}
\qquad
\frac{\vdash c_1 \Leftarrow t}{\vdash c_1 \vee c_2 \Leftarrow t}
\qquad
\frac{\vdash c_2 \Leftarrow t}{\vdash c_1 \vee c_2 \Leftarrow t}
\qquad
\frac{\vdash c_1 \Rightarrow t \qquad \vdash c_2 \Leftarrow s}{\vdash (c_1 \to c_2) \Leftarrow (t \to s)}$$

Figure 8: Contracts imply types

$$(\textbf{module}\ f\ e) \rhd (\textbf{module}\ f\ e)$$
$$(\textbf{module}\ f\ e) \rhd (\textbf{module}\ f\ c\ e)$$
$$\frac{e \rhd e'}{(\textbf{module}\ f\ e) \rhd (\textbf{module}\ f\ t\ e')}$$
$$x \rhd x \qquad f \rhd f \qquad n \rhd n$$
$$\frac{e \rhd e'}{e \rhd \{t \Leftarrow^f e'\}}$$
$$\frac{e_1 \rhd e_1' \qquad e_2 \rhd e_2'}{(e_1\ e_2) \rhd (e_1'\ e_2')}
\qquad
\frac{e \rhd e'}{(\lambda x.e) \rhd (\lambda x : t.e')}$$
$$\frac{e_1 \rhd e_1' \qquad e_2 \rhd e_2' \qquad e_3 \rhd e_3'}{(\textbf{if0}\ e_1\ e_2\ e_3) \rhd (\textbf{if0}\ e_1'\ e_2'\ e_3')}$$

Figure 9: Similarity

This captures our intuition as to how typed migration should work: that once we have migrated, we have proven the absence of errors in the typed module. Further, if we get an answer, it is related to the original answer. Since our reduction system tracks where errors occur, we are able to make this statement formally.

4.2 Soundness of our system

Proving soundness for our system is a multi-step process. First, we establish that the migrated system agrees with the original one, when errors are ignored. This is established through a simple relation between programs. Second, we define, and prove the correctness of, a transformation called ST that is simpler than MT. Finally, we prove that MT is appropriately related to ST.

For the first of these steps, we make use of the similarity property mentioned above and defined in figure 9. This relation between an untyped program (respectively, module or expression) and a partially typed one, states syntactically that the two programs P and P^M are similar, written $P \rhd P^M$, if they are identical ignoring contracts, types and casts.

Similarity satisfies three lemmas:

Lemma 1. *If $P \rhd P^M$ and $P \to^* w$ and $P^M \to^* w'$ then $w \rhd w'$.*

Lemma 2. *If $P \rhd P^M$ and P reduces forever then P^M reduces forever or $P^M \to^*$ $(\textbf{blame}\ f)$ for some f.*

Lemma 3. *If $P \rhd P^M$ and $P \to^*$ $(\textbf{blame}\ f)$ for some f then $P^M \to^*$ $(\textbf{blame}\ g)$ for some g.*

Proof Sketch These three lemmas all follow from similar bisimulation arguments. If $e_1 \rhd e_2$, then there are three possibilities:

1. $e_2 = E[\{c \Leftarrow v\}]$ Then either $e_2 \to (\textbf{blame}\ f)$ or $e_2 \to e_2'$ where $e_1 \rhd e_2'$. This can be seen by simple inspection of the reduction rules for casts.
2. $e_2 = E[(\{(c_1 \dashrightarrow c_2) \Leftarrow^f v\}\ w)]$. Then $e_2 \to e_2'$ and $e_1 \rhd e_2'$
3. $e_1 = E[r_1]$ and $e_2 = E[r_2]$ where $r_1 \rhd r_2$. Then $r_1 \to r_1'$ and $r_2 \to r_2'$ where $r_1' \rhd r_2'$ or $\rho_1 \to (\textbf{blame}\ f)$ and $r_2 \to (\textbf{blame}\ g)$. That the hypothesis holds is true from the definition of similarity and the grammar for $E[]$. The fact that the redexes reduce to similar terms or to errors can be seen from inspection of the reduction rules where the redex is not a cast or the application of a (blessed arrow) cast to a value.

Given this, similarity is consistently maintained by reduction, which is all we need for three three lemmas. $\square$

These lemmas are insufficient to establish soundness, since they make no claim about who is blamed for an error. To prove that the typed module is not the one blamed for errors, we introduce a different transformation on typed modules, for which it is possible to prove soundness with conventional techniques. This new transformation, ST, is defined as a sequence of four steps.

1. Choose one module M from P.
2. Add types to this module, so that is is now a typed module according to the grammar. Call this new module M'.
3. Apply the transformation and typechecking pass defined in figure 10 to the body of M'. That is, if
$$M' = (\textbf{module}\ \ M\ t\ e)$$
and $\Gamma \vdash^{ST} \emptyset : e; te'$ then
$$M'' = (\textbf{module}\ \ M\ t\ e')$$
4. Replace M in P with M'', producing P^M.

The transformation of figure 10 annotates every module reference in the typed module with a cast to the appropriate type. The key type rules are ST-ModVar, which adds a cast around a module reference, and ST-ModVarSelf, which handles self-reference to the typed module and does not insert a cast. In the program that results, no module has a contract. These invariants simplify the proof of soundness.

This transformation, like the original, is non-deterministic, thus $ST(P)$, like $MT(P)$, is a set. We must therefore prove that it is sound for *any* set of casts that it might generate.

Lemma 4 (Soundness of ST). *If $P^M \in ST(P)$ then $P \trianglerighteq P^M$.*

Proof Sketch By similarity, if $P \rightarrow^* v$ and $P^M \rightarrow^* v'$ then $v \triangleright v'$. Similarly, if P reduces forever, then $P^M \not\rightarrow^* v$ for any v. Therefore, the following lemma, stating that if an error occurs, the blame is assigned to one of the untyped modules, suffices for the proof. $\square$

Lemma 5 (ST never blames the typed module). *If $P^M \in ST(P)$, and $P^M \rightarrow^*$ (**blame** g) then $g \neq M$.*

Proof Sketch The only way we could ever reduce to (**blame** M) is if $(n\ v)^M$ is the redex or if a cast fails and blames M. To prove that neither of these happens, we show that the main expression is always *valid*, using the type system of Figure 11. Soundness is then implied by lemma 6. $\square$

Validity implies that there are no applications of numbers where the application is labeled with M, and also that every cast blaming M is applied to a term that satisfies that cast.

This type system, with judgement $\Gamma \vdash^M_{PM} e_m : t$ for mixed term e_m, allows us to type mixed terms even if they are not originally from the typed module M. This is key to the subsequent proofs, since we need to verify that both numbers and casted terms have the appropriate types, even if they arose from untyped sources.

There a several rules to note in the type system of figure 11. First, the rule T-Cast does not ensure that its argument is well-typed. Therefore, it applies even where the argument is an untyped term, and the cast is protecting the context of the cast from its argument. Second, the T-BlessedCast rule is necessary so that blessed casts can appear during reduction, even though they are not part of the syntax of types. Third, we allow the typed module to be used without a cast in rule T-TypeMod. Such module references are still protected from the untyped world, because they are within a typed expression.

We now define two important properties of mixed terms.

Definition 2. *A mixed term e is* consistent *in P^M iff $\emptyset \vdash^M_{PM} e : t$ for some type t.*

Terms may be consistent even if they do not originate in a typed module, or even if some of their subterms are not consistent. For example, $\emptyset \vdash^M_{PM} \{\textbf{int} \Leftarrow^f (\lambda x.(3\ x))\} : \textbf{int}$ for any f, even though the expression is patently erroneous.

Based on this definition, only some kinds of terms can be consistent: typed abstractions, numbers, casts and applications of two consistent terms.

Definition 3. *A mixed term e^f is* typed *in P^M iff f is the name of module M.*

This definition gives us a handle on those terms that came from the original typed module. These are the terms that must not trigger errors during the execution of the program.

With these definitions, we can define validity, the property that we use for our central lemma. This property ensures both that numbers are not in the operator position of a typed application, and that typed terms satisfy any immediately surrounding casts. Maintenance of these two properties is sufficient to ensure that M is never blamed. The third portion of the definition states that there is always a syntactic barrier between consistent and inconsistent portions of the expression, with the exception of numbers. This is the mechanism that is central to maintaining the other two during reduction.

Definition 4. *A mixed term e_m is* valid *in program P^M iff all of the following hold:*

1. *every typed subexpression $d^M \in e_m$ is either consistent or or the form $((\lambda x.e)\ e')^M$*
2. *for every expression of the form $\{t \Leftarrow^M e^M\}$, $\emptyset \vdash^M_{PM} e^M : t$.*
3. *every consistent term $d \in e_m$ is either a number or the immediate subterm of a consistent term*

Lemma 6. *If a mixed term e_m is valid in program $P^M \in ST(P)$ for some P, and $e_m \rightarrow e'_m$ then e'_m is valid in program P^M.*

Proof Sketch This proof proceeds by cases on the reduction rule that takes e_m to e'_m. The important cases are Split, Subst, TypedSubst and LookupTyped. We explain how to prove one of these here.

Consider case Split. Then $r = (\{(c_1 \dashrightarrow c_2) \Leftarrow^g v\}\ w)^f$ and $r' = \{c_2 \Leftarrow^g (v\ \{c_1 \Leftarrow^f w\})\}$. We consider the three components of validity in turn.

1. Since the casts and application in r' are new, they cannot be labeled M. Therefore, typed subexpressions of r' occur only in v and w. If v or w contain typed subexpression, these subexpressions must have been typed in r, and so by hypothesis they are still consistent. If the redex is part of a larger typed expression, then r must have been consistent, and must have had type c_2. But $\vdash^M_{PM} r' : c_2$, so this property is maintained.
2. The application expression in r' is new, and so cannot have label M. Therefore, we only need to consider

$$\frac{}{\Gamma \vdash^{ST} x : \Gamma(x); x}\ \text{ST-Var} \qquad \frac{}{\Gamma \vdash^{ST} f : t; \{t \Leftarrow^f f\}}\ \text{ST-ModVar} \qquad \frac{}{\Gamma \vdash^{ST} f : t; f \text{ if module } f \text{ has type } t \text{ in } P}\ \text{ST-ModVarSelf} \qquad \frac{}{\Gamma \vdash^{ST} n : \mathbf{int}; n}\ \text{ST-Int}$$

$$\text{ST-App}\ \frac{\Gamma \vdash^{ST} e_1 : (t_1 \to t_2); e_1' \qquad \Gamma \vdash^{ST} e_2 : t_1; e_2'}{\Gamma \vdash^{ST} (e_1\ e_2) : t_2; (e_1'\ e_2')} \qquad \text{ST-If0}\ \frac{\Gamma \vdash^{ST} e_1 : t_1; e_1' \qquad \Gamma \vdash^{ST} e_2 : t_2; e_2' \qquad \Gamma \vdash^{ST} e_3 : t_2; e_3'}{\Gamma \vdash^{ST} (\mathbf{if0}\ e_1\ e_2\ e_3) : t_2; (\mathbf{if0}\ e_1'\ e_2'\ e_3')}$$

$$\text{ST-Abs}\ \frac{\Gamma, x : t \vdash^{ST} e : s; e'}{\Gamma \vdash^{ST} (\lambda x : t.e) : (t \to s); (\lambda x : t.e')}$$

Figure 10: Simple Transformation

$$\text{T-Var}\ \Gamma \vdash^M_{PM} x : \Gamma(x) x \qquad \text{T-Cast}\ \Gamma \vdash^M_{PM} \{t \Leftarrow^f e\} : t \qquad \text{T-BlessedCast}\ \Gamma \vdash^M_{PM} \{(c_1 \dashrightarrow c_2) \Leftarrow^f v\} : (c_1 \to c_2) \qquad \text{T-Int}\ \Gamma \vdash^M_{PM} n : \mathbf{int}$$

$$\text{T-App}\ \frac{\Gamma \vdash^M_{PM} e_1 : (t_1 \to t_2) \qquad \Gamma \vdash^M_{PM} e_2 : t_1}{\Gamma \vdash^M_{PM} (e_1\ e_2) : t_2} \qquad \text{T-If0}\ \frac{\Gamma \vdash^M_{PM} e_1 : t_1 \qquad \Gamma \vdash^M_{PM} e_2 : t_2 \qquad \Gamma \vdash^M_{PM} e_3 : t_2}{\Gamma \vdash^M_{PM} (\mathbf{if0}\ e_1\ e_2\ e_3) : t_2} \qquad \text{T-Abs}\ \frac{\Gamma, x : t \vdash^M_{PM} e : s}{\Gamma \vdash^M_{PM} (\lambda x : t.e) : (t \to s)}$$

$$\text{T-TypeMod}\ \Gamma \vdash^M_{PM} f : t \text{ if } (\mathbf{module}\ \ M\ t\ e) \in P^M$$

Figure 11: Mixed Type System

the inner cast. If w is typed, then it must have been consistent (since the only other possibility is a redex). Therefore, the whole application must have been consistent, and thus $\vdash^M_{PM} w : c_1$, which is precisely the desired type.

3. Both casts trivially satisfy this case. Thus we have to consider v, w, and the application. Both the application and w are immediate arguments to a cast. If v is consistent, then it must be been a typed abstraction, since it is the argument of a blessed arrow contract, and untyped abstractions are not consistent. If it is a type-annotated abstraction, it must have label M, as required by the grammar. Thus, by hypothesis, it must satisfy its cast, and have type $(c_1 \to c_2)$. Therefore, since the operand is a cast to c_2, $\vdash^M_{PM} (v\ \{c_1 \Leftarrow^f w\}) : c_2$.

This concludes the case. The others are proved in a similar way. □

Given the soundness of ST, we can turn to proving soundness for MT. This relies on a relationship between contracts, as stated in the following lemma.

Lemma 7 (Soundness of the $\Rightarrow$ relation). *If* $\{t \Leftarrow^f v\} \to^* v'$ *and* $c \Rightarrow t$ *then* $\{c \Leftarrow^f v\} \to^* v'$.

Proof Sketch By induction on the derivation of $c \Rightarrow t$, either $c = t$ or c contains some disjunction in negative position, where t does not. If $c = t$ then the conclusion trivially follows. Further, by examination of the reduction rules Int-IntOr and Int-LamOr we note that if $\{c_1 \Leftarrow v\} \to v$ then $\{c_1 \vee c_2 \Leftarrow v\} \to v$. Therefore, these additional disjunctions will not introduce new failures that did not exist previously. □

With this, we can now conclude the main theorem of our paper.

Theorem 1 (Soundness of the transformation). *If* $P' \in MT(P)$ *then* $P \trianglerighteq P'$.

Proof Sketch Given the soundness of the simple transformation, all we need to prove is that every module variable reference that is not wrapped in a cast reduces to an appropriate value. By the definition of the MT transformation, however, every module reference is to a module to which we have added a contract. And by the lookup rule, that contract is turned into a cast at the point of reference. Therefore, by lemma 7, and the rules in figure 7 by which we add casts to the typed module, the new program still cannot blame the typed module. □

5. RELATED WORK

Over the past 20 years, researchers have made significant progress in related areas, both in typing untyped programs, and in inter-operation between languages with different type systems. Additionally, several systems have added a type discipline to previously untyped languages.

5.1 Soft Typing

Fagan and Cartwright [9], Aiken, Wimmers and Lakshman [2], Henglein and Rehof [15], Wright and Cartwright [30], Flanagan and Felleisen [12] and Meunier, Findler and Felleisen [23] studied the use of static analysis to infer types from untyped programs and to use the types to predict runtime errors statically in untyped programs. Meunier *et al*, whose calculus forms the basis for our own research, study an analysis that operated in the context of a first-order module system and a contract system. None of these systems considered the problem in a context with both typed and untyped code. Variants of these techniques will be useful, however, for automatically inserting the type annotations that we currently require programmers to write.

5.2 Interoperability

The problem of integrating typed languages with untyped ones has also seen significant study. Abadi, Cardelli, Pierce and Plotkin [1] considered the addition of a "type Dynamic" to a typed functional language. Values of "type Dynamic" can be created from arbitrary untyped sources such as I/O ports. The assumption is that the program is statically typed, and it periodically receives dynamically typed values from the outside world. The language requires an explicit typecase construct for deconstructing such values and converting them into some other type in the system. This approach is useful for a statically-typed language, but it does not form the basis for interlanguage migration. The programmer who wants to migrate a module from an untyped to a typed language must explicitly add typecase to every place the untyped code is used from the typed code, although this could potentially be done by a constraint analysis such as the one we present. More significantly, there is no provision for passing back and forth between the statically and the dynamically typed worlds. In contrast, our model allows full interaction between the two worlds. Systems based on "type Dynamic" have also been implemented and studied in the context of OCaml [18] and Haskell [4].

Gray, Findler and Flatt [13] consider the practical integration of a dynamically-typed language (Scheme) and a statically-typed one (Java), where back and forth communication is possible. Their work is in the spirit of extensive previous work on foreign function interfaces, but provides finer-grained interoperability, since Java and Scheme are both high-level languages. The work is presented in the context of interoperability, however, and does not address the issues involved in interlanguage migration.

Matthews and Findler [22], who continue where Gray *et al* leave off, discuss the meaning of programs that combine multiple languages. Their work concerns interoperability, however, not interlanguage migration. They do not attempt to generate general contracts that apply to the untyped modules under consideration. Instead, their system, in the spirit of traditional foreign function interfaces, requires programmers to insert syntactic separation between code in different languages. Requiring this in our system would require changes to the internals of modules other than the one being migrated, which is contrary to the spirit of modular development. Indeed, the source to the other modules may not even be accessible.[2]

Siek and Taha [26] propose "gradual typing", which incorporates both a type Dynamic with automatic coercions, as well as allowing only portions of types to be dynamic. The non-dynamic portions of types are then checked for consistency. This provides advantages over the traditional type Dynamic system, but does not address modularity, and does not provide a guarantee about where errors resulting from dynamic checks can occur.

Finally, there is extensive work on embedding typed and untyped languages on a single runtime system. For example, Jython, a version of Python [17], Groovy [14] and JScheme [3] are all untyped languages that run on the Java Virtual Machine [29], and several languages, include both Python [16] and Scheme [10] have also been ported to Microsoft's .NET runtime. Since the runtime in these cases is statically typed, the implementers of these languages are forced to embed all of their values into a single host-language type. Additionally, numerous strongly typed languages are able to interact with C, which is not type-safe. While these systems are related, none addresses the central issues of our work: support for the interlanguage migration process and of the soundness of that migration. In fact, where interlanguage migration is done using such systems, it is often in the direction of less safety, in order to achieve greater performance.

[2]While our transformation does add contracts to existing modules, this only requires changes to the interface, not to the internal implementation.

5.3 Explicit Types for Dynamic Languages

Several previous systems focus on adding a type system to an existing untyped language. The Strongtalk system [7] is a type checker for Smalltalk code that relies on programmer annotations for types, but erases all types and runs the underlying Smalltalk code at execution time. Strongtalk would therefore be a candidate for use in our migration framework. Although the Strongtalk system did integrate with the (untyped) Smalltalk class library, no provision is made for ensuring type safety in the presence of untyped code. Indeed, it is unclear what guarantees are made by the Strongtalk type system.

Similarly, the Cecil type system of Litvinov [19] and the Erlang type system of Marlow and Wadler [21] also can be ignored at runtime, but again do not deal with the integration between typed and untyped code. Recently, Bracha [6] proposes adding multiple optional type systems to dynamic languages, but does not address the details of the interactions between such systems.

Numerous compilers for dynamic languages attempt to infer the types left unstated by the programmer for performance reasons. A number of Lisp and Scheme systems go further and allow type declarations, which can be used as a guide to these optimizations [28, 25]. The Python compiler for Common Lisp [20] ensures that these declarations hold via static and dynamic checking, but it does not attempt to enforce a type safety property.

6. CONCLUSION

In this paper we have presented the first framework for an interlanguage migration process and a technique for proving it correct. The framework identifies important steps of the process: typing modules; deriving constraints on the rest of the program from the types; and turning the constraints into behavioral contracts for module interfaces. For our simple concrete instance, we have also been able to show that interlanguage migration truly adds type safety in the expected manner: the typed portion of the program can't go wrong; only the untyped portions may trigger run-time exceptions.

Working out the the framework suggests several pieces of future work. First, interlanguage migration needs tools that help programmers rewrite modules so that they satisfy a type discipline. This requires simultaneous work on more powerful type systems and analyses such as soft-typing and static debugging. Second, we need to study how much type safety we can get with richer type systems than the simply-typed λ-calculus.

Given the explosive use of dynamically typed scripting languages and the growing desire of programmers to improve the quality of their programs, we believe that interlanguage migration will become an important element in the software engineering process. This paper is only a first step in the exploration of this process; much more work needs to be done to find a truly satisfying framework for a relatively clean scripting language such as Scheme.

Acknowledgments Jacob Matthews' NU PRL presentation last fall inspired our use of contracts to mediate between the untyped modules and the newly typed module. Dave Herman, Shriram Krishnamurthi, Phillipe Meunier and the anonymous referees of DLS suggested many improvements to an early version of the paper.

7. REFERENCES

[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, 1991.

[2] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1994. ACM Press.

[3] Ken Anderson, Timothy Hickey, and Peter Norvig. JScheme user manual, 2002. http://jscheme.sf.net.

[4] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166, New York, NY, USA, 2002. ACM Press.

[5] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley Professional, November 2002.

[6] Gilad Bracha. Pluggable type systems. In *Revival of Dynamic Languages workshop at OOPSLA 2004*, October 2004.

[7] Gilad Bracha and David Griswold. Strongtalk: typechecking smalltalk in a production environment. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, New York, NY, USA, 1993. ACM Press.

[8] Frederick P. Brooks Jr. *The Mythical Man-Month.* Addison Wesley, 1995. Anniversary Edition.

[9] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

[10] William D. Clinger. Common Larceny. In *International Lisp Conference*, pages 101–107, June 2005. Invited paper.

[11] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, New York, NY, USA, 2002. ACM Press.

[12] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, 1999.

[13] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 231–245, New York, NY, USA, 2005. ACM Press.

[14] Groovy, 2004–2006. http://groovy.codehaus.org.

[15] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: translating Scheme to ML. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 192–203, New York, NY, USA, 1995. ACM Press.

[16] Jim Hugunin. IronPython, 2003–2006. http://www.ironpython.com.

[17] Jython, 2003–2006. http://www.jython.com.

[18] Xavier Leroy and Michel Mauny. Dynamics in ML. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 406–426, London, UK, 1991. Springer-Verlag.

[19] Vassily Litvinov. Contraint-based polymorphism in Cecil: towards a practical and static type system. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 388–411, New York, NY, USA, 1998. ACM Press.

[20] Robert A. MacLachlan. The Python compiler for CMU Common Lisp. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 235–246, New York, NY, USA, 1992. ACM Press.

[21] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 136–149, New York, NY, USA, 1997. ACM Press.

[22] Jacob Matthews and Robert Bruce Findler. The Meaning of Multi-Language Programs, 2006. Unpublished manuscript.

[23] Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 218–231, New York, NY, USA, 2006. ACM Press.

[24] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer Systems Science*, 17:348–375, 1978.

[25] Manuel Serrano. *Bigloo: A practical Scheme compiler*, 1992–2002.

[26] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, 2006. ACM Press.

[27] Dorai Sitaram. ScheLog, http://www.ccs.neu.edu/home/dorai/schelog/schelog.html 1993–2004.

[28] Guy Lewis Steele Jr. *Common Lisp—The Language.* Digital Press, 1984.

[29] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (Second Edition).* Sun Microsystems, 1999.

[30] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. *ACM Trans. Program. Lang. Syst.*, 19(1):87–152, 1997.