Higher-Order Symb Comput (2010) 23:167–189
DOI 10.1007/s10990-011-9066-z

Space-efficient gradual typing

David Herman · Aaron Tomb · Cormac Flanagan

Published online: 21 October 2011

Abstract Gradual type systems offer a smooth continuum between static and dynamic typing by permitting the free mixture of typed and untyped code. The runtime systems for these languages, and other languages with hybrid type checking, typically enforce function types by dynamically generating function proxies. This approach can result in unbounded growth in the number of proxies, however, which drastically impacts space efficiency and destroys tail recursion.

We present a semantics for gradual typing that is based on *coercions* instead of function proxies, and which combines adjacent coercions at runtime to limit their space consumption. We prove bounds on the space consumed by coercions as well as soundness of the type system, demonstrating that programmers can safely mix typing disciplines without incurring unreasonable overheads. Our approach also detects certain errors earlier than prior work.

Keywords Gradual typing · Coercions · Casts · Type dynamic

David Herman was supported by a grant from the Mozilla Corporation. Cormac Flanagan was supported by a Sloan Fellowship and by NSF grant CCS-0341179. Aaron Tomb was also supported by NSF grant CCS-0341179. For Herman, most of this work was done at Northeastern University, Boston, MA.

D. Herman (✉)
Mozilla Research, 650 Castro St, Mountain View, CA 94116, USA
e-mail: dherman@mozilla.com

A. Tomb · C. Flanagan
Computer Science Department, School of Engineering, University of California at Santa Cruz, 1156 High Street, Santa Cruz, CA 95064, USA

A. Tomb
e-mail: atomb@soe.ucsc.edu

C. Flanagan
e-mail: cormac@ucsc.edu

1 Gradual typing for software evolution

Dynamically typed languages have always excelled at exploratory programming. Languages such as Lisp, Scheme, Smalltalk, and JavaScript support quick early prototyping and incremental development without the overhead of documenting (often-changing) structural invariants as types. For large applications, however, static type systems have proven invaluable. They are crucial for understanding and enforcing key program invariants and abstractions, and they catch many errors early in the development process.

Given these different strengths, it is not uncommon to encounter the following scenario: a programmer builds a prototype in a dynamically-typed scripting language, perhaps even in parallel to a separate, official software development process with an entire team. The team effort gets mired in process issues and over-engineering, and the prototype ends up in production use. Before long, this hastily conceived prototype grows into a full-fledged production system, but without the structure or guarantees provided by static types. The system becomes unwieldy and bugs are cropping up faster than the team can reproduce them. Ultimately, they decide to port the application to a statically typed language, requiring a complete rewrite of the entire system.

The cost of cross-language migration is huge and often insupportable. But the scenario above is avoidable. Several languages combine static and dynamic typing, among them Boo [11], Visual Basic.NET [26], Sage [20], Clean [5, 29], and Racket [34, 35]. This approach of *hybrid typing*, where types are enforced with a combination of static and dynamic checks, has begun to gain attention in the research community [4, 8, 20, 26, 32, 34, 35]. Recently, Siek and Taha [32, 33] coined the slogan *gradual typing* for this important application of hybrid typing: the ability to implement both partially-conceived prototypes and mature, production systems in the same programming language by gradually introducing type discipline. Gradual typing offers the possibility of continuous software evolution from prototype to product, thus avoiding the huge costs of language migration.

Our experience in the technical committee working on the JavaScript [13] language specification provides a more concrete example. JavaScript is a dynamically-typed, higher-order programming language and a key enabling technology for interactive web applications. JavaScript is used both for very simple scripts and for mature, large-scale applications. The enormous popularity of the language is due in no small part to its low barrier to entry; anyone can write a JavaScript program by copying and pasting code from one web page to another. Its dynamic type system and fail-soft runtime semantics allow programmers to produce useful programs with a minimum of effort. The increasing complexity of modern web applications, however, has motivated the desire for more robust mechanisms to support programming in the large. In past years, the technical committee considered the design of a hybrid type system for JavaScript, which would allow the mixing of dynamic and static typing disciplines, to support gradual typing [12]. Gradual typing is still a relatively new research area, however, and so the technical committee would prefer further experimental validation before considering incorporating gradual typing into JavaScript.

This paper extends our earlier conference paper [23] with additional context and perspective on the relationship of our contributions with prior and subsequent work on hybrid and gradual typing.

1.1 The cost of gradual typing

The interaction between typed and untyped code imposes certain requirements on the design of the runtime semantics of gradually-typed languages. Gradually-typed languages support

both statically-typed and dynamically-typed code, and include runtime checks (or type casts) at the boundaries between these two typing disciplines, to guarantee that dynamically-typed code cannot violate the invariants of statically-typed code.

To illustrate this idea, consider the following code fragment, which passes a dynamically-typed variable x into a variable y of type `Int`:

$$\begin{array}{l}\texttt{let}\ x : \texttt{Dyn} = \texttt{true}\ \texttt{in} \\ \texttt{let}\ y : \texttt{Int} = x\ \texttt{in} \\ \ldots\end{array}$$

During compilation, the type checker inserts a dynamic type cast $\langle\texttt{Int}\rangle$ to enforce the type invariant on y, yielding the "compiled" code:

$$\begin{array}{l}\texttt{let}\ x : \texttt{Dyn} = \texttt{true}\ \texttt{in} \\ \texttt{let}\ y : \texttt{Int} = \langle\texttt{Int}\rangle\ x\ \texttt{in} \\ \ldots\end{array}$$

At runtime, this $\langle\texttt{Int}\rangle$ cast detects the attempted type violation, since x contains `true`:

$$\begin{array}{ll} & \texttt{let}\ x : \texttt{Dyn} = \texttt{true}\ \texttt{in}\ \texttt{let}\ y : \texttt{Int} = \langle\texttt{Int}\rangle\ x\ \texttt{in}\ \ldots \\ \longrightarrow & \texttt{let}\ y : \texttt{Int} = \langle\texttt{Int}\rangle\ \texttt{true}\ \texttt{in}\ \ldots \\ \longrightarrow & \textsf{Error} : \textit{"failed cast"}\end{array}$$

Unfortunately, even these simple, first-order type checks can impose unexpected costs, as in the following example, where a programmer has added some type annotations to a previously untyped program:

$$\textit{even} : \texttt{Dyn} \to \texttt{Dyn} \stackrel{\text{def}}{=} \lambda n{:}\texttt{Dyn}.\ \textsf{if}\ (n = 0)\ \textsf{then}\ \texttt{true}\ \ \textsf{else}\ \textit{odd}\ (n - 1)$$

$$\textit{odd} : \texttt{Int} \to \texttt{Bool} \stackrel{\text{def}}{=} \lambda n{:}\texttt{Int}.\ \textsf{if}\ (n = 0)\ \textsf{then}\ \texttt{false}\ \textsf{else}\ \textit{even}\ (n - 1)$$

This program seems innocuous, but suffers from a subtle space leak. Since *even* is dynamically typed, the result of each call to *even* must be implicitly cast to `Bool`. That is, after the type checker inserts the necessary casts, the compiled version $\textit{odd}_c$ of *odd* is:

$$\begin{array}{l}\textit{odd}_c : \texttt{Int} \to \texttt{Bool} \stackrel{\text{def}}{=} \lambda n{:}\texttt{Int}.\ \textsf{if}\ (n = 0) \\ \qquad\qquad \textsf{then}\ \texttt{false}\ \textsf{else}\ \langle\texttt{Bool}\rangle\ (\textit{even}\ (\langle\texttt{Dyn}\rangle\ (n - 1)))\end{array}$$

These implicit casts result in unbounded growth in the control stack and so destroy tail recursion.

Additional complications arise when first-class functions cross the boundaries between typing disciplines. In general, it is not possible to check if an untyped function satisfies a particular static type. A natural solution is to wrap the function in a proxy that, whenever it is applied, casts its argument and result values appropriately, ensuring that the function is only observed with its expected type. This proxy-based approach was formalized by Findler and Felleisen [15] and widely used in subsequent literature [16, 20, 25, 32], but it has serious consequences for space efficiency.

As a simple example, consider the following program in continuation-passing style, where both mutually recursive functions take a continuation argument k, but only one of

these arguments is annotated with a precise type:

$$\begin{aligned} \mathit{evenk} : \mathtt{Int} \to \mathtt{Bool} \quad &\stackrel{\text{def}}{=} \quad \lambda n{:}\mathtt{Int}.\, \lambda k{:}(\mathtt{Dyn} \to \mathtt{Dyn}). \\ &\qquad \mathsf{if}\ (n = 0)\ \mathsf{then}\ (k\ \mathtt{true})\ \ \mathsf{else}\ \mathit{oddk}\ (n-1)\ k \end{aligned}$$

$$\begin{aligned} \mathit{oddk} : \mathtt{Int} \to \mathtt{Bool} \quad &\stackrel{\text{def}}{=} \quad \lambda n{:}\mathtt{Int}.\, \lambda k{:}(\mathtt{Bool} \to \mathtt{Bool}). \\ &\qquad \mathsf{if}\ (n = 0)\ \mathsf{then}\ (k\ \mathtt{false})\ \mathsf{else}\ \mathit{evenk}\ (n-1)\ k \end{aligned}$$

During compilation, the type checker inserts a dynamic type cast on the continuation argument k, yielding the compiled code:

$$\begin{aligned} \mathit{evenk}_c : \mathtt{Int} \to \mathtt{Bool} \quad &\stackrel{\text{def}}{=} \quad \lambda n{:}\mathtt{Int}.\, \lambda k{:}(\mathtt{Dyn} \to \mathtt{Dyn}). \\ &\qquad \mathsf{if}\ (n = 0) \\ &\qquad \mathsf{then}\ \langle \mathtt{Bool} \rangle\ (k\ (\langle \mathtt{Dyn} \rangle\ \mathtt{true})) \\ &\qquad \mathsf{else}\ \mathit{oddk}_c\ (n-1)\ (\langle \mathtt{Bool} \to \mathtt{Bool} \rangle\ k) \end{aligned}$$

$$\begin{aligned} \mathit{oddk}_c : \mathtt{Int} \to \mathtt{Bool} \quad &\stackrel{\text{def}}{=} \quad \lambda n{:}\mathtt{Int}.\, \lambda k{:}(\mathtt{Bool} \to \mathtt{Bool}). \\ &\qquad \mathsf{if}\ (n = 0) \\ &\qquad \mathsf{then}\ (k\ \mathtt{false}) \\ &\qquad \mathsf{else}\ \mathit{evenk}_c\ (n-1)\ (\langle \mathtt{Dyn} \to \mathtt{Dyn} \rangle\ k) \end{aligned}$$

Applying these functions to a sufficiently large n reveals the impact on space consumption of performing repeated function casts to k:

$$\begin{aligned} & \mathit{evenk}_c\ \ n\ \ (\langle \mathtt{Dyn} \to \mathtt{Dyn} \rangle\ (\lambda x{:}\mathtt{Bool}.\, x)) \\ \longrightarrow^* & \mathit{evenk}_c\ \ n\ \ (\lambda z_1{:}\mathtt{Dyn}.\, \langle \mathtt{Dyn} \rangle\ ((\lambda x{:}\mathtt{Bool}.\, x)\ (\langle \mathtt{Bool} \rangle\ z_1))) \\ \longrightarrow^* & \mathit{oddk}_c\ \ (n-1)\ \ (\langle \mathtt{Bool} \to \mathtt{Bool} \rangle\ (\lambda z_1{:}\mathtt{Dyn}.\, \langle \mathtt{Dyn} \rangle\ ((\lambda x{:}\mathtt{Bool}.\, x)\ (\langle \mathtt{Bool} \rangle\ z_1)))) \\ \longrightarrow^* & \mathit{oddk}_c\ \ (n-1)\ \ (\lambda z_2{:}\mathtt{Bool}.\, \langle \mathtt{Bool} \rangle \\ & \qquad\qquad (\lambda z_1{:}\mathtt{Dyn}.\, \langle \mathtt{Dyn} \rangle\ ((\lambda x{:}\mathtt{Bool}.\, x)\ (\langle \mathtt{Bool} \rangle\ z_1)) \\ & \qquad\qquad\ (\langle \mathtt{Dyn} \rangle\ z_2)) \\ \longrightarrow^* & \mathit{evenk}_c\ (n-2)\ \ (\langle \mathtt{Dyn} \to \mathtt{Dyn} \rangle \\ & \qquad\qquad \lambda z_2{:}\mathtt{Bool}.\, \langle \mathtt{Bool} \rangle \\ & \qquad\qquad\quad (\lambda z_1{:}\mathtt{Dyn}.\, \langle \mathtt{Dyn} \rangle\ ((\lambda x{:}\mathtt{Bool}.\, x)\ (\langle \mathtt{Bool} \rangle\ z_1)) \\ & \qquad\qquad\quad\ (\langle \mathtt{Dyn} \rangle\ z_2))) \\ \longrightarrow^* & \mathit{evenk}_c\ (n-2)\ \ (\lambda z_3{:}\mathtt{Dyn}.\, \langle \mathtt{Dyn} \rangle \\ & \qquad\qquad (\lambda z_2{:}\mathtt{Bool}.\, \langle \mathtt{Bool} \rangle \\ & \qquad\qquad\quad (\lambda z_1{:}\mathtt{Dyn}.\, \langle \mathtt{Dyn} \rangle\ ((\lambda x{:}\mathtt{Bool}.\, x)\ (\langle \mathtt{Bool} \rangle\ z_1)) \\ & \qquad\qquad\quad\ (\langle \mathtt{Dyn} \rangle\ z_2)) \\ & \qquad\qquad\ (\langle \mathtt{Bool} \rangle\ z_3))) \\ \longrightarrow^* & \ldots \end{aligned}$$

The recursive calls to oddk_c and evenk_c quietly cast the continuation argument k with higher-order casts $\langle \mathtt{Bool} \to \mathtt{Bool} \rangle$ and $\langle \mathtt{Dyn} \to \mathtt{Dyn} \rangle$, respectively. This means that the function argument k is wrapped in an additional function proxy at each recursive call! Despite the fact that the function argument is only used at a total of two different types, its size grows in proportion to the depth of the recursion, due to these dynamically generated proxies.

These examples demonstrate that naïve gradual typing violates space safety of functional programming languages by incurring surprising and unnecessary changes in asymptotic space consumption of programs. The purpose of gradual typing is to support the seamless integration of typing disciplines, especially for the common use case of simple scripts

that mature into large applications. Subtle, significant, and pervasive costs in resource consumption threaten to limit the utility of gradual typing for its intended use.

Of course, if *even* and *odd* are in the same module or maintained by the same development team, a natural solution is to avoid mixing typing disciplines, and to consistently use static typing (or dynamic typing) for both functions. A more problematic situation, however, occurs at interfaces between different modules maintained by different teams, as for example between a GUI library that uses callbacks to a GUI application. In this situation, we might want to migrate the GUI library to be more statically typed, while still maintaining backward compatibility with pre-existing GUI applications.

In short, the flexibility promised by gradual typing can only be achieved if programmers are free to decide how precisely to type various parts of a program. This flexibility is lost if adding type annotations can accidentally trigger drastic changes in its memory usage. The goal of this paper is to combine the flexibility and ease-of-use of gradual type systems with predictable resource consumption.

1.2 Space-efficient gradual typing using coercions

We present a semantics for gradual typing that overcomes these problems. Our approach hinges on the simple insight that when proxies accumulate at runtime, they often contain redundant information. In the higher-order example above, the growing chain of function proxies contains only two distinct components, $\texttt{Bool} \to \texttt{Bool}$ and $\texttt{Dyn} \to \texttt{Dyn}$, which could be merged to the simpler but equivalent $\texttt{Bool} \to \texttt{Bool}$.

Type casts behave like *error projections* [14], which are closed under composition. However, the syntax of casts does not always compose; for example, there is no cast c such that $\langle c \rangle\ e = \langle \texttt{Int} \rangle\ \langle \texttt{Bool} \rangle\ e$. Furthermore, projections are idempotent, which should allow us to eliminate duplicate casts. For example, the following equality holds in a semantic sense:

$$\langle \texttt{Bool} \to \texttt{Bool} \rangle\ \langle \texttt{Bool} \to \texttt{Bool} \rangle\ e = \langle \texttt{Bool} \to \texttt{Bool} \rangle\ e$$

But such transformations are inconvenient, if not impossible, with a representation of higher-order type casts as functions.

Our formalization instead leverages Henglein's *coercion calculus* [22], which provides a syntax for projections, called coercions, which are closed under a composition operator. This allows us to combine adjacent casts whenever they occur at runtime in order to eliminate redundant information and thus guarantee clear bounds on space consumption. The details of combining and simplifying coercions are presented in Sect. 6. By eagerly combining coercions, we can also detect certain errors immediately as soon as a function cast is applied; in contrast, prior approaches would not detect these errors until the casted function is invoked.

Our approach is applicable to many gradually- or hybrid-typed languages [16, 17, 19, 20] that use function proxies and hence are prone to space consumption problems. For clarity, we formalize our approach for the simply-typed λ-calculus with references. Of course, gradual typing is not restricted to simple type systems: the Sage language [20] incorporates gradual typing as part of a very expressive type system with polymorphic functions, type operators, first-class types, dependent types, general refinement types, etc. Concurrently, Siek and Taha [32] developed gradual typing for the simpler language $\lambda^{?}_{\to}$, which we use as the basis for this presentation.

1.3 Early error detection

A second contribution of this work is that it permits early detection of errors, and hence better test coverage. To illustrate this idea, consider the following code fragment, which erroneously attempts to convert an (`Int` → `Int`) function to have type (`Bool` → `Int`):

$$\texttt{let}\ g : \texttt{Dyn} = (\lambda x : \texttt{Int}.\ x)\ \texttt{in}$$
$$\texttt{let}\ h : (\texttt{Bool} \to \texttt{Int}) = g\ \texttt{in}\ \ldots$$

Most prior strategies for gradual typing would not detect this error until h is applied to a boolean argument, causing the integer cast to fail.

In contrast, our coercion-based implementation allows us to detect this error as soon as h is defined, which detects gradual typing errors earlier and in more situations. We believe that the eager approach better fits the philosophy of gradual typing, which is to permit maximum flexibility while still detecting errors as early as possible.

1.4 Outline

The presentation of our results proceeds as follows. The following section provides some perspective on related work done prior to and following the original publication of this work. Section 3 reviews the syntax and type system of $\lambda^{?}_{\to}$. Section 4 introduces the coercion algebra underlying our approach. Section 5 describes how we translate source programs into a target language with explicit coercions. Section 6 provides an operational semantics for that target language. Section 7 extends our approach to detect errors earlier and in more situations. Section 8 proves bounds on the space consumption. The last two sections discuss other items of related work and place our results into context.

2 Context and perspective

This paper is an extended version of an earlier conference paper [23] that provides additional context and perspective by relating our contributions with both prior and subsequent work in the general area of hybrid and gradual typing.

Figure 1 summarizes some of the central characteristics of a few selected recent papers in this area. These papers vary in regards to whether they support each of the following features:

- precise blame tracking;
- the dynamic type `Dyn`;
- predicate types, such as $\{\mathtt{x} : \mathtt{Int} \mid \mathtt{x} \geq 0\}$, which denotes non-negative integers;
- early detection of nonsensically-composed function casts at run-time, before the function is applied; and
- guarantees of space efficiency.

The final column in Fig. 1 describes the representation of function casts, either as: functions $(\lambda x.\cdots)$; as casts of the form $\langle S \Leftarrow T \rangle$ converting from type T to type S; as Henglein's coercions; or as *Threesomes* [30]. We discuss these papers and characterizations in more detail below.

All of these papers leverage and extend the general technique of *higher-order casts* formalized by Findler and Felleisen [15]. This technique dynamically enforces contracts on

Paper	Higher Order Casts	Blame	`Dyn` Type	Predicate Types	Early Error Checks	Space Efficient	Cast Represent.
Contract Systems:							
Findler and Felleisen [15]	X	X					Function
Hybrid Type Systems:							
Flanagan [16]	X			X			Function
Knowles and Flanagan [24]	X			X			Function
Gronski et al. [20]	X		X	X			Function
Greenberg et al. [19]	X	X		X			Cast
Gradual Type Systems:							
Siek and Taha [32, 33]	X		X				Cast
Herman et al. [23]	X		X		X	X	Coercion
Siek et al. [31]	X	X	X		X	X	Coercion
Siek and Wadler [30]	X	X	X		X	X	Threesome

Fig. 1 A comparison of hybrid and gradual typing system

functions by creating wrapper function *proxies*, which check that the appropriate contracts hold on all function arguments and results, in a lazy manner. A key contribution of this work was a mechanism for tracking blame through these function wrappers and for assigning blame for any contract failure back to a particular contract in the source program. This technique of higher-order wrappers (both with and without blame) was subsequently applied in a number of domains.

Hybrid type checking uses higher-order casts to overcome decidability and expressiveness limitations of traditional static type systems. The initial presentation [16] developed these ideas for an extension of the simply-typed lambda calculus with *predicate types*, where a base type such as `Int` is refined by an arbitrary boolean expression, to yield a predicate type such as $\{\mathtt{x} : \mathtt{Int} \mid \mathtt{x} \geq 0\}$. In combination with dependent function types, predicate types permit the specification of rather precise function interfaces, which facilitate modular software development. Ou et al. [28] earlier explored similar ideas for interoperation between precisely and coarsely typed code. Subsequent work added a variety of extensions, including a dynamic type `Dyn` [20], clarifications to the metatheory [24], and extensions to support blame tracking and other properties [19].

Concurrently, Siek and Taha used higher-order wrappers to permit convenient interoperation between statically- and dynamically-typed code, targeting both functional [32] and object-oriented [33] languages. The omission of predicate types resulted in a significantly simpler language semantics and metatheory.

This paper (and our earlier conference paper [23]) introduces and satisfies two additional desiderata. Most critical is space efficiency, without which gradually-typed programs can be prone to unexpected blow-ups in stack and heap space requirements. An additional desideratum is early detection of errors, which was lacking in several earlier papers. We leverage Henglein's coercions [22] to represent higher-order casts at run-time, which facilitates cast normalization for space efficiency.

Subsequently, Siek and Taha [31] successfully extended these ideas with blame tracking, while still preserving the eager error detection and space efficiency properties. In addition, their paper also clarified several other aspects of the design space for gradual typing systems.

Henglein's coercions provide a convenient run-time representation for higher-order casts, since sequences of coercions can be combined or normalized at runtime, which ensures the desired space efficiency guarantee. However, coercions are a rather low-level and indirect representation of the casts that originated in the source program. To avoid this indirect representation, Siek and Taha [30] introduced the notion of a *threesome* as a replacement for a normalized sequence of coercions. In particular, a threesome $\langle T_3 \Leftarrow T_2 \Leftarrow T_1 \rangle$ is essentially a downcast from a type T_1 to T_2 followed by an upcast from T_2 to T_3. Note that this threesome is not equivalent to a conventional type cast $\langle T_3 \Leftarrow T_1 \rangle$ from T_1 to T_3, since the intermediate downcast to T_2 may fail on more argument values. This threesome representation corresponds to normalized coercions, and so can achieve similar space efficiency guarantees while still reasoning at the level of types and casts, rather than using lower-level coercions.

Figure 1 provides an overall comparison between the features supported by various hybrid and gradual typed systems. In general, gradual typing avoids the complex metatheory associated with predicate types—this simpler setting has allowed gradual type systems to innovate and explore in other directions, namely space efficiency and early error detection.

We suggest that a fruitful avenue for future research is to merge these two closely related research directions by developing a flexible and expressive type system that supports static, dynamic, and predicate types, with a space-efficient run-time that tracks blame and performs early error checks. We suspect that an appropriate strategy is to appropriately extend the threesome cast representation to support predicate types. Additional directions for future work are discussed in Sect. 10.

3 Gradually-typed lambda calculus

This section reviews the gradually-typed λ-calculus $\lambda^{?}_{\rightarrow}$ that we use to formalize our ideas. This language is essentially the simply-typed λ-calculus extended with the type `Dyn` to represent dynamic types; it also includes mutable reference cells to demonstrate the gradual typing of assignments. We refer the interested reader to Siek and Taha's work [32] for motivations regarding the design of this language and its type system.

$$
\begin{array}{rrll}
\textit{Typing environments:} & E & ::= & \emptyset \mid E, x : T \\
\textit{Terms:} & e & ::= & k \mid x \mid \lambda x{:}T.\, e \mid e\ e \mid \texttt{ref}\ e \mid\ !e \mid e := e \\
\textit{Types:} & S, T & ::= & B \mid T \rightarrow T \mid \texttt{Dyn} \mid \texttt{Ref}\ T
\end{array}
$$

Terms include the usual constants, variables, abstractions, and applications, as well as reference allocation, dereference, and assignment. Types include the dynamic type `Dyn`, function types $T \rightarrow T$, reference types `Ref` T, and some collection of ground or base types B (such as `Int` or `Float`).

The $\lambda^{?}_{\rightarrow}$ type system is a little unusual in that it is based on an intransitive *consistency* relation $S \sim T$ instead of the more conventional transitive subtyping relation $S <: T$. Any type is consistent with the type `Dyn`, from which it follows that, for example, `Bool` $\sim$ `Dyn` and `Dyn` $\sim$ `Int`. However, booleans cannot be used directly as integers, which is why the consistency relation is not transitively closed. We do not assume the consistency relation is symmetric, since our language allows coercions from integers to floats but not vice-versa.

The consistency relation is defined in Fig. 2. Rules [C-DynL] and [C-DynR] allow all coercions to and from type `Dyn`. The rule [C-Float] serves as an example of asymmetry by

Consistency rules $\boxed{S \sim T}$

$$\frac{}{T \sim T}\ [\text{C-Refl}] \qquad \frac{}{T \sim \texttt{Dyn}}\ [\text{C-DynR}] \qquad \frac{}{\texttt{Dyn} \sim T}\ [\text{C-DynL}] \qquad \frac{}{\texttt{Int} \sim \texttt{Float}}\ [\text{C-Float}]$$

$$\frac{T_1 \sim S_1 \qquad S_2 \sim T_2}{(S_1 \to S_2) \sim (T_1 \to T_2)}\ [\text{C-Fun}] \qquad \frac{T \sim S \qquad S \sim T}{\texttt{Ref}\ S \sim \texttt{Ref}\ T}\ [\text{C-Ref}]$$

Type rules $\boxed{E \vdash e : T}$

$$\frac{(x : T) \in E}{E \vdash x : T}\ [\text{T-Var}] \qquad \frac{E, x : S \vdash e : T}{E \vdash (\lambda x{:}S.\ e) : (S \to T)}\ [\text{T-Fun}] \qquad \frac{}{E \vdash k : ty(k)}\ [\text{T-Const}]$$

$$\frac{E \vdash e_1 : (S \to T) \qquad E \vdash e_2 : S' \qquad S' \sim S}{E \vdash (e_1\ e_2) : T}\ [\text{T-App1}] \qquad \frac{E \vdash e_1 : \texttt{Dyn} \qquad E \vdash e_2 : S}{E \vdash (e_1\ e_2) : \texttt{Dyn}}\ [\text{T-App2}]$$

$$\frac{E \vdash e : T}{E \vdash \texttt{ref}\ e : \texttt{Ref}\ T}\ [\text{T-Ref}] \qquad \frac{E \vdash e : \texttt{Ref}\ T}{E \vdash !e : T}\ [\text{T-Deref1}] \qquad \frac{E \vdash e : \texttt{Dyn}}{E \vdash !e : \texttt{Dyn}}\ [\text{T-Deref2}]$$

$$\frac{E \vdash e_1 : \texttt{Ref}\ T \qquad E \vdash e_2 : S \qquad S \sim T}{E \vdash e_1 := e_2 : S}\ [\text{T-Assign1}] \qquad \frac{E \vdash e_1 : \texttt{Dyn} \qquad E \vdash e_2 : T}{E \vdash e_1 := e_2 : \texttt{Dyn}}\ [\text{T-Assign2}]$$

Fig. 2 Source language type system

allowing coercion from `Int` to `Float` but not the reverse. The rule [C-Fun] is reminiscent of the contravariant/covariant rule for function subtyping. We extend the invariant reference cells of $\lambda^{?}_{\to}$ to allow coercion from `Ref` S to `Ref` T via rule [C-Ref], provided S and T are symmetrically consistent. Unlike functions, reference cells do not distinguish their output ("read") type from their input ("write") type, so coercion must be possible in either direction. For example, the two reference types `Ref Int` and `Ref Dyn` are consistent.

Figure 2 also presents the type rules for the source language. Notice the presence of two separate procedure application rules. Rule [T-App1] checks statically-typed functions; in this case, the argument may have any type consistent with the function's domain. Rule [T-App2] handles the case where the operator is dynamically-typed; in this case the argument may be of any type. The assignment rules follow an analogous pattern, accepting a consistent type when the left-hand side is known to have type `Ref` T, and any type when the left-hand side is dynamically-typed. Similarly, dereferences only produce precise types for reference types.

4 Coercions

To achieve a space-efficient implementation, we elaborate source programs into a target language with explicit type casts, which allow expressions of one type to be used at any consis-

tent type. Our representation of casts is based on *coercions*, drawn from Henglein's theory of dynamic typing [22]. The key benefit of coercions over prior proxy-based representations is that they are *combinable*; if two coercions are wrapped around a function value, then they can be safely combined into a single coercion, thus reducing the space consumption of the program without changing its semantic behavior.

The coercion language and its typing rules are both defined in Fig. 3. The coercion judgment $\vdash c : S \rightsquigarrow T$ states that coercion c serves to coerce values from type S to type T (but may fail on some values of type S that are not coerceable to type T). The identity coercion I always succeeds, and has type $\vdash c : T \rightsquigarrow T$ for any T. Conversely, the failure coercion `Fail` always fails, but can relate any two types S and T, and so $\vdash \texttt{Fail} : S \rightsquigarrow T$. For each base type B, the coercion $B?$ "downcasts" a value of type `Dyn` to type B, and so has type $\vdash B? : \texttt{Dyn} \rightsquigarrow B$. Conversely, the tagging coercion $B!$ has type $\vdash B! : B \rightsquigarrow \texttt{Dyn}$. Similarly, the function checking coercion `Fun?` converts a value of type `Dyn` to have the dynamic function type $\texttt{Dyn} \to \texttt{Dyn}$. If a more precise function type is required, this value can be further coerced via a function coercion $\texttt{Fun}\ c\ d$, where c coerces function arguments and d coerces results. For example, the coercion (`Fun Int? Int!`) coerces from $\texttt{Dyn} \to \texttt{Dyn}$ to $\texttt{Int} \to \texttt{Int}$, by untagging function arguments (via `Int?`) and tagging function results (via `Int!`). Conversely, the tagging coercion `Fun!` coerces a dynamic function of type $\texttt{Dyn} \to \texttt{Dyn}$ to type `Dyn`. Like function coercions, reference coercions also contain two components: the first for coercing values put into the reference cell; the second for coercing values read from the cell. Finally, the coercion $c; d$ represents coercion composition, *i.e.*, the coercion c followed by coercion d.

This coercion language is sufficient to translate between all consistent types: if types S and T are consistent, then the following partial function $\mathit{coerce}(S, T)$ is defined and returns the appropriate coercion between these types.

$$\begin{aligned}
\mathit{coerce} &: \mathit{Type} \times \mathit{Type} \to \mathit{Coercion} \\
\mathit{coerce}(T, T) &= I \\
\mathit{coerce}(B, \texttt{Dyn}) &= B! \\
\mathit{coerce}(\texttt{Dyn}, B) &= B? \\
\mathit{coerce}(\texttt{Int}, \texttt{Float}) &= \texttt{IntFloat} \\
\mathit{coerce}(S_1 \to S_2, T_1 \to T_2) &= \texttt{Fun}\ \mathit{coerce}(T_1, S_1)\ \mathit{coerce}(S_2, T_2) \\
\mathit{coerce}(\texttt{Dyn}, T_1 \to T_2) &= \texttt{Fun?}; \mathit{coerce}(\texttt{Dyn} \to \texttt{Dyn}, T_1 \to T_2) \\
\mathit{coerce}(T_1 \to T_2, \texttt{Dyn}) &= \mathit{coerce}(T_1 \to T_2, \texttt{Dyn} \to \texttt{Dyn}); \texttt{Fun!} \\
\mathit{coerce}(\texttt{Ref}\ S, \texttt{Ref}\ T) &= \texttt{Ref}\ \mathit{coerce}(T, S)\ \mathit{coerce}(S, T) \\
\mathit{coerce}(\texttt{Dyn}, \texttt{Ref}\ T) &= \texttt{Ref?}; \mathit{coerce}(\texttt{Ref Dyn}, \texttt{Ref}\ T) \\
\mathit{coerce}(\texttt{Ref}\ T, \texttt{Dyn}) &= \mathit{coerce}(\texttt{Ref}\ T, \texttt{Ref Dyn}); \texttt{Ref!}
\end{aligned}$$

Coercing a type T to itself produces the identity coercion I. Coercing base types B to type `Dyn` requires a tagging coercion $B!$, and coercing `Dyn` to a base type B requires a runtime check $B?$. Function coercions work by coercing their domain and range types. The type `Dyn` is coerced to a function type via a two-step coercion: first the value is checked to be a function and then coerced from the dynamic function type $\texttt{Dyn} \to \texttt{Dyn}$ to $T_1 \to T_2$. Dually, typed functions are coerced to type `Dyn` via coercion to a dynamic function type followed by the function tag `Fun!`. Coercing a $\texttt{Ref}\ S$ to a $\texttt{Ref}\ T$ entails coercing all writes from T to S and all reads from S to T. Coercing reference types to and from `Dyn` is analogous to function coercion.

Syntax

$$\begin{array}{rrcl} \textit{Coercions:} & c, d & ::= & I \mid \texttt{Fail} \mid G! \mid G? \mid \texttt{IntFloat} \mid \texttt{Fun}\ c\ c \mid \texttt{Ref}\ c\ c \mid c; c \\ \textit{Dynamic tags:} & G & ::= & B \mid \texttt{Fun} \mid \texttt{Ref} \end{array}$$

Coercion rules $\boxed{\vdash c : S \rightsquigarrow T}$

$$\frac{}{\vdash I : T \rightsquigarrow T}\ [\text{C-Id}] \qquad \frac{}{\vdash \texttt{Fail} : S \rightsquigarrow T}\ [\text{C-Fail}] \qquad \frac{}{\vdash B! : B \rightsquigarrow \texttt{Dyn}}\ [\text{C-}B!] \qquad \frac{}{\vdash B? : \texttt{Dyn} \rightsquigarrow B}\ [\text{C-}B?]$$

$$\frac{}{\vdash \texttt{Fun!} : (\texttt{Dyn} \to \texttt{Dyn}) \rightsquigarrow \texttt{Dyn}}\ [\text{C-Fun!}] \qquad \frac{}{\vdash \texttt{Fun?} : \texttt{Dyn} \rightsquigarrow (\texttt{Dyn} \to \texttt{Dyn})}\ [\text{C-Fun?}]$$

$$\frac{\vdash c_1 : T_1' \rightsquigarrow T_1 \qquad \vdash c_2 : T_2 \rightsquigarrow T_2'}{\vdash (\texttt{Fun}\ c_1\ c_2) : (T_1 \to T_2) \rightsquigarrow (T_1' \to T_2')}\ [\text{C-Fun}]$$

$$\frac{}{\vdash \texttt{Ref!} : (\texttt{Ref Dyn}) \rightsquigarrow \texttt{Dyn}}\ [\text{C-Ref!}] \qquad \frac{}{\vdash \texttt{Ref?} : \texttt{Dyn} \rightsquigarrow (\texttt{Ref Dyn})}\ [\text{C-Ref?}]$$

$$\frac{\vdash c : T \rightsquigarrow S \qquad \vdash d : S \rightsquigarrow T}{\vdash (\texttt{Ref}\ c\ d) : (\texttt{Ref}\ S) \rightsquigarrow (\texttt{Ref}\ T)}\ [\text{C-Ref}]$$

$$\frac{\vdash c_1 : T \rightsquigarrow T_1 \qquad \vdash c_2 : T_1 \rightsquigarrow T_2}{\vdash (c_1; c_2) : T \rightsquigarrow T_2}\ [\text{C-Compose}] \qquad \frac{}{\vdash \texttt{IntFloat} : \texttt{Int} \rightsquigarrow \texttt{Float}}\ [\text{C-Float}]$$

Fig. 3 Coercion language and type rules

Lemma 1 (Well-typed coercions)

1. $S \sim T$ *iff* $\mathit{coerce}(S, T)$ *is defined.*
2. *If* $c = \mathit{coerce}(S, T)$ *then* $\vdash c : S \rightsquigarrow T$.

Proof Inductions on the derivations of $S \sim T$ and $\mathit{coerce}(S, T)$. □

Example Consider again the continuation-passing definition of *evenk* in Sect. 1.1, which passes its continuation argument k to *oddk*. Because *oddk* expects a continuation argument of type $\texttt{Bool} \to \texttt{Bool}$ but k has type $\texttt{Dyn} \to \texttt{Dyn}$, the type checker must insert a cast around the use of k. Note that the types $\texttt{Bool} \to \texttt{Bool}$ and $\texttt{Dyn} \to \texttt{Dyn}$ are consistent, from which it follows by Lemma 1 that *coerce* is defined on these arguments:

$$\begin{aligned} &\mathit{coerce}(\texttt{Dyn} \to \texttt{Dyn}, \texttt{Bool} \to \texttt{Bool}) \\ &= \texttt{Fun}\ \mathit{coerce}(\texttt{Bool}, \texttt{Dyn})\ \mathit{coerce}(\texttt{Dyn}, \texttt{Bool}) \\ &= \texttt{Fun Bool! Bool?} \end{aligned}$$

Syntax

$$
\begin{array}{rrcl}
\textit{Terms:} & s,t & ::= & x \mid u \mid t\ t \mid \texttt{ref}\ t \mid !t \mid t := t \mid \langle c\rangle\ t \\
\textit{Values:} & v & ::= & u \mid \langle vc\rangle\ u \\
\textit{Uncoerced values:} & u & ::= & \lambda x{:}T.\ t \mid k \mid a \\
\textit{Value coercions:} & vc & ::= & G! \mid \texttt{Fun}\ c\ d \mid \texttt{Ref}\ c\ d \mid vc; c \\
\textit{Stores:} & \sigma & ::= & \emptyset \mid \sigma[a := v] \\
\textit{Typing environments:} & E & ::= & \emptyset \mid E, x : T \\
\textit{Store typings:} & \Sigma & ::= & \emptyset \mid \Sigma, a : T
\end{array}
$$

Type rules for terms $\boxed{E; \Sigma \vdash t : T}$

$$\text{[T-Var]}\ \frac{(x:t) \in E}{E;\Sigma \vdash x : T} \qquad \text{[T-Fun]}\ \frac{E, x:S;\Sigma \vdash t : T}{E;\Sigma \vdash (\lambda x{:}S.\ t) : (S \to T)} \qquad \text{[T-App]}\ \frac{E;\Sigma \vdash t_1 : (S \to T) \qquad E;\Sigma \vdash t_2 : S}{E;\Sigma \vdash (t_1\ t_2) : T}$$

$$\text{[T-Ref]}\ \frac{E;\Sigma \vdash t : T}{E;\Sigma \vdash \texttt{ref}\ t : \texttt{Ref}\ T} \qquad \text{[T-Deref]}\ \frac{E;\Sigma \vdash t : \texttt{Ref}\ T}{E;\Sigma \vdash !t : T} \qquad \text{[T-Assign]}\ \frac{E;\Sigma \vdash t_1 : \texttt{Ref}\ T \qquad E;\Sigma \vdash t_2 : T}{E;\Sigma \vdash t_1 := t_2 : T}$$

$$\text{[T-Const]}\ \frac{}{E;\Sigma \vdash k : ty(k)} \qquad \text{[T-Cast]}\ \frac{\vdash c : S \rightsquigarrow T \qquad E;\Sigma \vdash t : S}{E;\Sigma \vdash \langle c\rangle\ t : T} \qquad \text{[T-Addr]}\ \frac{(a:T) \in \Sigma}{E;\Sigma \vdash a : \texttt{Ref}\ T}$$

Type rules for stores $\boxed{E; \Sigma \vdash \sigma}$

$$\text{[T-Store]}\ \frac{dom(\sigma) = dom(\Sigma) \qquad \forall a \in dom(\sigma).\ E;\Sigma \vdash \sigma(a) : \Sigma(a)}{E;\Sigma \vdash \sigma}$$

Fig. 4 Target language syntax and type rules

Furthermore, the resulting coercion has the expected type according to the coercion rules of Fig. 3:

$$\frac{\vdash \texttt{Bool!} : \texttt{Bool} \rightsquigarrow \texttt{Dyn} \qquad \vdash \texttt{Bool?} : \texttt{Dyn} \rightsquigarrow \texttt{Bool}}{\vdash (\texttt{Fun Bool! Bool?}) : (\texttt{Dyn} \to \texttt{Dyn}) \rightsquigarrow (\texttt{Bool} \to \texttt{Bool})}$$

5 Target language and cast insertion

During compilation, we simultaneously type-check the source program and insert explicit type casts to convert between consistent types where necessary. The target language of this cast insertion process is essentially the same as the source, except that it uses explicit casts of the form $\langle c\rangle\ t$ as the only mechanism for connecting terms of type `Dyn` and terms of other types. For example, the term $\langle \texttt{Int?}\rangle\ x$ has type `Int`, provided that x has type `Dyn`. The

Cast insertion rules $\boxed{E \vdash e \hookrightarrow t : T}$

$$\frac{(x : T) \in E}{E \vdash x \hookrightarrow x : T}\ [\text{C-VAR}] \qquad \frac{}{E \vdash k \hookrightarrow k : ty(k)}\ [\text{C-CONST}] \qquad \frac{E, x : S \vdash e \hookrightarrow t : T}{E \vdash (\lambda x{:}S.\ e) \hookrightarrow (\lambda x{:}S.\ t) : (S \to T)}\ [\text{C-FUN}]$$

$$\frac{E \vdash e_1 \hookrightarrow t_1 : (S \to T) \qquad E \vdash e_2 \hookrightarrow t_2 : S' \qquad c = coerce(S', S)}{E \vdash e_1\ e_2 \hookrightarrow (t_1\ (\langle c \rangle\ t_2)) : T}\ [\text{C-APP1}]$$

$$\frac{E \vdash e_1 \hookrightarrow t_1 : \texttt{Dyn} \qquad E \vdash e_2 \hookrightarrow t_2 : S' \qquad c = coerce(S', \texttt{Dyn})}{E \vdash e_1\ e_2 \hookrightarrow ((\langle \texttt{Fun?} \rangle\ t_1)\ (\langle c \rangle\ t_2)) : \texttt{Dyn}}\ [\text{C-APP2}]$$

$$\frac{E \vdash e \hookrightarrow t : T}{E \vdash \texttt{ref}\ e \hookrightarrow \texttt{ref}\ t : \texttt{Ref}\ T}\ [\text{C-REF}] \qquad \frac{E \vdash e \hookrightarrow t : \texttt{Ref}\ T}{E \vdash !e \hookrightarrow !t : T}\ [\text{C-DEREF1}] \qquad \frac{E \vdash e \hookrightarrow t : \texttt{Dyn}}{E \vdash !e \hookrightarrow !(\langle \texttt{Ref?} \rangle\ t) : \texttt{Dyn}}\ [\text{C-DEREF2}]$$

$$\frac{E \vdash e_1 \hookrightarrow t_1 : \texttt{Ref}\ S \qquad E \vdash e_2 \hookrightarrow t_2 : T \qquad c = coerce(T, S)}{E \vdash e_1 := e_2 \hookrightarrow (t_1 := (\langle c \rangle\ t_2)) : S}\ [\text{C-ASSIGN1}]$$

$$\frac{E \vdash e_1 \hookrightarrow t_1 : \texttt{Dyn} \qquad E \vdash e_2 \hookrightarrow t_2 : T \qquad c = coerce(T, \texttt{Dyn})}{E \vdash e_1 := e_2 \hookrightarrow ((\langle \texttt{Ref?} \rangle\ t_1) := (\langle c \rangle\ t_2)) : \texttt{Dyn}}\ [\text{C-ASSIGN2}]$$

Fig. 5 Cast insertion rules

language syntax and type rules are defined in Fig. 4, and are mostly straightforward. The values include "uncoerced values" u, consisting of abstractions, constants, and references. The latter are represented as addresses a in the global store σ. A value may also be wrapped with a *value coercion* vc. Value coercions are the subset of coercions that begin with either a tag $G!$ or a higher-order coercion $\texttt{Fun}\ c\ d$ or $\texttt{Ref}\ c\ d$.

The process of type checking and inserting coercions is formalized via the *cast insertion judgment*:

$$E \vdash e \hookrightarrow t : T$$

Here, the type environment E provides types for free variables, e is the original source program, t is a modified version of the original program with additional coercions, and T is the inferred type for t. The rules defining the cast insertion judgment are shown in Fig. 5, and they rely on the partial function *coerce* to compute coercions between types. For example, rule [C-APP1] compiles an application expression where the operator has a function type $S \to T$ by casting the argument expression to type S. Rule [C-APP2] handles the case where the operator is dynamically-typed by inserting a $\langle \texttt{Fun?} \rangle$ check to ensure the operator evaluates to a function, and casting the argument to type $\texttt{Dyn}$ to yield a tagged value. Rules [C-REF] and [C-DEREF1] handle typed reference allocation and dereference. Rule [C-DEREF2] handles dynamically-typed dereference by inserting a runtime $\langle \texttt{Ref?} \rangle$ check. Rule [C-ASSIGN1] handles statically-typed assignment, casting the right-hand side to the expected type of the reference cell, and [C-ASSIGN2] handles dynamically-typed assignment, casting the left-hand side with a runtime $\langle \texttt{Ref?} \rangle$ check and the right-hand side with a tagging coercion to type $\texttt{Dyn}$.

Returning to the continuation-passing *evenk* and *oddk* functions, the cast insertion judgment produces the following compiled program (which differs somewhat from the compiled version in the introduction because of our use of coercions to represent casts):

$$\begin{aligned} \mathit{evenk}_c : \mathtt{Int} \to \mathtt{Bool} \stackrel{\text{def}}{=}\ & \lambda n{:}\mathtt{Int}.\ \lambda k{:}(\mathtt{Dyn} \to \mathtt{Dyn}). \\ & \quad \mathsf{if}\ (n = 0) \\ & \quad \mathsf{then}\ \langle \mathtt{Bool?} \rangle\ (k\ (\langle \mathtt{Bool!} \rangle\ \mathtt{true})) \\ & \quad \mathsf{else}\ \mathit{oddk}_c\ (n-1)\ (\langle \mathtt{Fun\ Bool!\ Bool?} \rangle\ k) \end{aligned}$$

$$\begin{aligned} \mathit{oddk}_c : \mathtt{Int} \to \mathtt{Bool} \stackrel{\text{def}}{=}\ & \lambda n{:}\mathtt{Int}.\ \lambda k{:}(\mathtt{Bool} \to \mathtt{Bool}). \\ & \quad \mathsf{if}\ (n = 0) \\ & \quad \mathsf{then}\ (k\ \mathtt{false}) \\ & \quad \mathsf{else}\ \mathit{evenk}_c\ (n-1)\ (\langle \mathtt{Fun\ Bool?\ Bool!} \rangle\ k) \end{aligned}$$

Compilation succeeds on all well-typed source programs, and produces only well-typed target programs.

Theorem 1 (Well-typed programs compile) *For all* E, e, *and* T, *the following statements are equivalent*:

1. $E \vdash e : T$
2. $\exists t$ *such that* $E \vdash e \hookrightarrow t : T$

Theorem 2 (Compiled programs are well-typed) *For all* E, e, t, *and* T, *if* $E \vdash e \hookrightarrow t : T$ *then* $E; \emptyset \vdash t : T$

Proof Inductions on the cast insertion and source language typing derivations. □

6 Operational semantics

We now consider how to implement the target language in a manner that limits the space consumed by coercions. The key idea is to combine adjacent casts at runtime to eliminate redundant information while preserving the behavior of programs.

Figure 6 provides the definitions and reduction rules for the target language, using a small-step operational semantics with evaluation contexts. The grammar of evaluation contexts C is defined in terms of contexts D, which cannot begin with a cast. This prevents nesting of adjacent casts and allows the crucial [E-CCAST] reduction rule to ensure that adjacent casts in the program term are always merged.

In order to maintain bounds on their size, coercions are maintained normalized throughout evaluation according to the following rules:

$$\begin{array}{rr} I; c = c & G!; G? = I \\ c; I = c & G!; G'? = \mathtt{Fail} \quad \text{if } G \neq G' \\ \mathtt{Fail}; c = \mathtt{Fail} & (\mathtt{Fun}\ c_1\ c_2); (\mathtt{Fun}\ d_1\ d_2) = \mathtt{Fun}\ (d_1; c_1)\ (c_2; d_2) \\ c; \mathtt{Fail} = \mathtt{Fail} & (\mathtt{Ref}\ c_1\ c_2); (\mathtt{Ref}\ d_1\ d_2) = \mathtt{Ref}\ (d_1; c_1)\ (c_2; d_2) \\ & (c_1; c_2); c_3 = c_1; (c_2; c_3) \end{array}$$

This normalization is applied in a transitive, compatible manner whenever the rule [E-CCAST] is applied, thus bounding the size of coercions during evaluation. Normalized

Syntax

$$\begin{array}{rrcl} \textit{Evaluation Contexts:} & C & ::= & D \mid \langle c\rangle\ D \\ \textit{Non-cast Contexts:} & D & ::= & \bullet \mid (C\ t) \mid (v\ C) \mid \texttt{ref}\ C \mid\ !C \mid C := t \mid v := C \end{array}$$

Evaluation rules

$$\begin{array}{rcll} C[(\lambda x{:}S.\ t)\ v], \sigma & \longrightarrow & C[t[x := v]], \sigma & & \text{[E-Beta]} \\ C[\texttt{ref}\ v], \sigma & \longrightarrow & C[a], \sigma[a := v] & \text{for } a \notin dom(\sigma) & \text{[E-New]} \\ C[!a], \sigma & \longrightarrow & C[\sigma(a)], \sigma & \text{if } a \in dom(\sigma) & \text{[E-Deref]} \\ C[a := v], \sigma & \longrightarrow & C[v], \sigma[a := v] & \text{if } a \in dom(\sigma) & \text{[E-Assign]} \\ C[k\ v], \sigma & \longrightarrow & C[\delta(k, v)], \sigma & & \text{[E-Prim]} \\ C[(\langle \texttt{Fun}\ c\ d\rangle\ u)\ v], \sigma & \longrightarrow & C[\langle d\rangle\ (u\ (\langle c\rangle\ v))], \sigma & & \text{[E-CBeta]} \\ C[!(\langle \texttt{Ref}\ c\ d\rangle\ a)], \sigma & \longrightarrow & C[\langle d\rangle\ !a], \sigma & & \text{[E-CDeref]} \\ C[(\langle \texttt{Ref}\ c\ d\rangle\ a) := v], \sigma & \longrightarrow & C[\langle d\rangle\ (a := \langle c\rangle\ v)], \sigma & & \text{[E-CAssign]} \\ C[\langle I\rangle\ u], \sigma & \longrightarrow & C[u], \sigma & \text{if } C \neq C'[\langle c\rangle\ \bullet] & \text{[E-Id]} \\ C[\langle \texttt{IntFloat}\rangle\ n], \sigma & \longrightarrow & C[\mathit{nearestFloat}(n)], \sigma & \text{if } C \neq C'[\langle c\rangle\ \bullet] & \text{[E-FCast]} \\ C[\langle c\rangle\ (\langle d\rangle\ t)], \sigma & \longrightarrow & C[\langle nc\rangle\ t], \sigma & \text{if } d;c = nc & \text{[E-CCast]} \end{array}$$

Fig. 6 Operational semantics

coercions satisfy the following grammar (where [—] denotes an optional component):

$$\begin{array}{rcl} nc & ::= & I \mid \texttt{Fail} \mid G? \mid G! \mid G?; G! \\ & \mid & [\texttt{Int?};]\,\texttt{IntFloat}\,[;\texttt{Float!}] \\ & \mid & [\texttt{Fun?};]\,\texttt{Fun}\ nc\ nc\,[;\texttt{Fun!}] \\ & \mid & [\texttt{Ref?};]\,\texttt{Ref}\ nc\ nc\,[;\texttt{Ref!}] \end{array}$$

Most of the remaining reduction rules are straightforward. Rules [E-Beta], [E-New], [E-Deref], and [E-Assign] are standard. The rule [E-Prim] relies on a function $\delta : \mathit{Term} \times \mathit{Term} \to \mathit{Term}$ to define the semantics of constant functions. For simplicity, we assume each δ to be defined for all constants k and arguments v. Rule [E-CBeta] applies function casts by casting the function argument and result. Rule [E-CDeref] casts the result of reading a cell and [E-CAssign] casts the value written to a cell and casts the value again to the expected output type. Rules [E-Id] and [E-FCast] respectively perform the identity and float coercions, and are restricted to non-cast contexts to prevent overlap with [E-CCast].

6.1 Space-efficient coercions

Using these evaluation rules, the continuation-passing $evenk_c$ and $oddk_c$ functions of Sect. 1.1 now maintain normalized coercions during evaluation:

$$\begin{array}{lll} & \mathit{evenk}_c\ \ n & (\langle \texttt{Fun Bool? Bool!}\rangle\ \lambda x{:}\texttt{Bool}.\ x) \\ \longrightarrow^* & \mathit{oddk}_c\ (n-1) & (\langle \texttt{Fun Bool! Bool?}\rangle\ \langle \texttt{Fun Bool? Bool!}\rangle\ \lambda x{:}\texttt{Bool}.\ x) \\ \longrightarrow & \mathit{oddk}_c\ (n-1) & (\langle (\texttt{Fun Bool? Bool!});\ (\texttt{Fun Bool! Bool?})\rangle\ \lambda x{:}\texttt{Bool}.\ x) \qquad \text{[E-CCast]} \\ = & \mathit{oddk}_c\ (n-1) & (\langle \texttt{Fun (Bool!; Bool?) (Bool!; Bool?)}\rangle\ \lambda x{:}\texttt{Bool}.\ x) \\ = & \mathit{oddk}_c\ (n-1) & (\langle \texttt{Fun}\ I\ I\rangle\ \lambda x{:}\texttt{Bool}.\ x) \\ \longrightarrow^* & \mathit{evenk}_c\ (n-2) & (\langle \texttt{Fun Bool? Bool!}\rangle\ \langle \texttt{Fun}\ I\ I\rangle\ \lambda x{:}\texttt{Bool}.\ x) \\ = & \mathit{evenk}_c\ (n-2) & (\langle \texttt{Fun Bool? Bool!}\rangle\ \lambda x{:}\texttt{Bool}.\ x) \\ \longrightarrow^* & \ldots & \end{array}$$

Note the crucial role played by rule [E-CCAST] in this reduction sequence to combine the adjacent function casts. Once these casts are combined, the coercion normalization rules simplify them down to (Fun I I) at each subsequent invocation of $oddk_c$.

$$\begin{aligned} & (\texttt{Fun Bool? Bool!}); (\texttt{Fun Bool! Bool?}) \\ = \; & \texttt{Fun (Bool!; Bool?) (Bool!; Bool?)} \\ = \; & \texttt{Fun } I \; I \end{aligned}$$

6.2 Tail recursion

Consider the compiled version of the direct-style *even* and *odd* functions:

$$\begin{aligned} even_c : \texttt{Dyn} \to \texttt{Dyn} \quad &\stackrel{\text{def}}{=} \quad \lambda n{:}\texttt{Dyn}. \\ &\qquad \textsf{if } (\langle \texttt{Int?} \rangle \; n) = 0 \\ &\qquad \textsf{then } \langle \texttt{Bool!} \rangle \; \texttt{true} \\ &\qquad \textsf{else } \langle \texttt{Bool!} \rangle \; (odd_c \; ((\langle \texttt{Int?} \rangle \; n) - 1))) \end{aligned}$$

$$\begin{aligned} odd_c : \texttt{Int} \to \texttt{Bool} \quad &\stackrel{\text{def}}{=} \quad \lambda n{:}\texttt{Int}. \\ &\qquad \textsf{if } (n = 0) \\ &\qquad \textsf{then } \texttt{false} \\ &\qquad \textsf{else } \langle \texttt{Bool?} \rangle \; (even_c \; (\langle \texttt{Int!} \rangle \; (n - 1))) \end{aligned}$$

In the following evaluation sequence, the definition of evaluation contexts forces the evaluation of rule [E-CCAST] before reducing the application of *odd*. In other words, tail calls are evaluated only *after* normalizing the cast at the top of the program stack. Thus, as shown below, there is no unbounded growth in the control stack.

$$\begin{aligned} & \langle \texttt{Bool?} \rangle \; (even_c \; (\langle \texttt{Int!} \rangle \; 4)) \\ \longrightarrow^* \; & \langle \texttt{Bool?} \rangle \; \langle \texttt{Bool!} \rangle \; (odd_c \; 3) \\ \longrightarrow \; & \langle \texttt{Bool!; Bool?} \rangle \; (odd_c \; 3) \\ = \; & \langle I \rangle \; (odd_c \; 3) \\ \longrightarrow^* \; & \langle \texttt{Bool?} \rangle \; (even_c \; (\langle \texttt{Int!} \rangle \; 2)) \\ \longrightarrow^* \; & \ldots \end{aligned}$$

6.3 Type soundness

Evaluation satisfies the usual preservation and progress lemmas.

Theorem 3 (Soundness of evaluation) *If* $\emptyset; \emptyset \vdash t : T$ *then either*

1. $t, \emptyset$ *diverges*,
2. $t, \emptyset \longrightarrow^* C[\langle \texttt{Fail} \rangle \; u], \sigma$ *or*
3. $t, \emptyset \longrightarrow^* v, \sigma$ *and* $\exists \Sigma$ *such that* $\emptyset; \Sigma \vdash v : T$ *and* $\emptyset; \Sigma \vdash \sigma$.

Proof Via standard subject reduction and progress lemmas in the style of Wright and Felleisen [37]. □

7 Early error detection

Our coercion-based implementation of casts allows the runtime to detect errors earlier in the evaluation sequence and in more situations than earlier systems for gradual typing. To illustrate this idea, consider the following code fragment, which erroneously attempts to convert an $(\texttt{Int} \to \texttt{Int})$ function to have type $(\texttt{Bool} \to \texttt{Int})$:

$$\begin{array}{l}\texttt{let } f : (\texttt{Int} \to \texttt{Int}) = (\lambda x : \texttt{Int}.\ x) \texttt{ in} \\ \texttt{let } h : (\texttt{Bool} \to \texttt{Int}) = f \texttt{ in } \ldots\end{array}$$

Most gradual type systems (including ours) will detect this error at compile time, since these types are not consistent: $(\texttt{Int} \to \texttt{Int}) \not\sim (\texttt{Bool} \to \texttt{Int})$. However, the introduction of an additional variable of type Dyn into the data-flow path hides this error from the type system, since $(\texttt{Int} \to \texttt{Int}) \sim \texttt{Dyn} \sim (\texttt{Bool} \to \texttt{Int})$:

$$\begin{array}{l}\texttt{let } f : (\texttt{Int} \to \texttt{Int}) = (\lambda x : \texttt{Int}.\ x) \texttt{ in} \\ \texttt{let } g : \texttt{Dyn} = f \texttt{ in} \\ \texttt{let } h : (\texttt{Bool} \to \texttt{Int}) = g \texttt{ in } \ldots\end{array}$$

For this revised example, the function representation of higher-order casts would not detect this error until h is applied to a boolean argument, causing the integer cast to fail.

In contrast, our coercion-based implementation allows us to detect this error as soon as h is initialized. After cast insertion and evaluation, the value of h is

$$\begin{array}{l}h = \langle(\texttt{Fun Bool! Int?}); (\texttt{Fun Int? Int!})\rangle\ (\lambda x : \texttt{Int}.\ x) \\ \quad = \langle\texttt{Fun } (\texttt{Int?}; \texttt{Bool!})\ (\texttt{Int?}; \texttt{Int!})\rangle\ (\lambda x : \texttt{Int}.\ x) \\ \quad = \langle\texttt{Fun Fail } I\rangle\ (\lambda x : \texttt{Int}.\ x)\end{array}$$

where the domain coercion Fail explicates the inconsistency of the two domain types Int and Bool. We can modify our semantics to halt as soon as such inconsistencies are detected, by adding the following coercion normalization rules to lift this failure to the top level:

$$\begin{array}{ll}\texttt{Fun } c\ \texttt{Fail} = \texttt{Fail} & \texttt{Ref } c\ \texttt{Fail} = \texttt{Fail} \\ \texttt{Fun Fail } c = \texttt{Fail} & \texttt{Ref Fail } c = \texttt{Fail}\end{array}$$

Under these additional normalization rules, we have:

$$\begin{array}{l}h = \langle\texttt{Fun Fail } I\rangle\ (\lambda x : \texttt{Int}.\ x) \\ \quad = \langle\texttt{Fail}\rangle\ (\lambda x : \texttt{Int}.\ x)\end{array}$$

which is not a value, and so evaluation gets stuck.

Using these rules, our implementation strategy halts as soon as h is initialized, resulting in earlier runtime error detection and the detection of errors in more situations than under the function representation of higher-order casts. We believe that the eager approach better fits the philosophy of gradual typing, which is to permit maximum flexibility while still detecting errors as early as possible.

8 Space efficiency

We now consider how much space coercions consume at runtime, beginning with an analysis of how much space each individual coercion can consume.

8.1 Space consumption

The size of a coercion $size(c)$ is defined as the size of its abstract syntax tree representation. When two coercions are sequentially composed and normalized during evaluation, the size of the normalized, composed coercion may of course be larger than either of the original coercions. In order to reason about the space required by such composed coercions, we introduce a notion of the *height* of a coercion:

$$height(I) = height(\texttt{Fail}) = height(G!) = height(G?) = 1$$
$$height(\texttt{Ref}\ c\ d) = height(\texttt{Fun}\ c\ d) = 1 + max(height(c), height(d))$$
$$height(c; d) = max(height(c), height(d))$$

Notably, the height of a composed coercion is bounded by the maximum height of its constituents. In addition, normalization never increases the height of a coercion. Thus, the height of any coercion created during program evaluation is never larger than the height of some coercion in the original elaborated program.

Furthermore, this bound on the height of each coercion in turn guarantees a bound on the coercion's size, according to the following lemma. In particular, even though the length of a coercion sequence $c_1; \ldots; c_n$ does not contribute to its height, the restricted structure of well-typed, normalized coercions constrains the length (and hence size) of such sequences.

Lemma 2 *For all well-typed normalized coercions* c, $size(c) \leq 5(2^{height(c)} - 1)$.

Proof Induction on c. Assume $c = c_1; \ldots; c_n$, where each c_i is not a sequential composition.

Suppose some $c_i = \texttt{Ref}\ d_1\ d_2$. So $\vdash c_i : \texttt{Ref}\ S \rightsquigarrow \texttt{Ref}\ T$. Hence c_i can be preceded only by Ref?, which must be the first coercion in the sequence, and similarly can be followed only by Ref!, which must be the last coercion in the sequence. Thus in the worst case $c = \texttt{Ref?}; \texttt{Ref}\ d_1\ d_2; \texttt{Ref!}$ and $size(c) = 5 + size(d_1) + size(d_2)$. Applying the induction hypothesis to the sizes of d_1 and d_2 yields:

$$size(c) \leq 5 + 2(5(2^{height(c)-1} - 1)) = 5(2^{height(c)} - 1)$$

The case for $\texttt{Fun}\ d_1\ d_2$ is similar. The coercions I and Fail can only appear alone. Finally, coercions of the form $G?; G!$ are valid. However, composition of a coercion c matching this pattern with one of the other valid coercions is either ill-typed or triggers a normalization that yields a coercion identical to c. □

In addition, the height of any coercion created during cast insertion is bounded by the height of some type in the typing derivation (where the height of a type is the height of its abstract syntax tree representation).

Lemma 3 *If* $c = coerce(S, T)$, *then* $height(c) \leq max(height(S), height(T))$.

Proof Induction on the structure of $coerce(S, T)$. □

Theorem 4 (Size of coercions) *For any* e_0, c *such that*

1. $\emptyset \vdash e_0 \hookrightarrow t_0 : T$ *and*
2. $t_0, \emptyset \longrightarrow^* t, \sigma$ *and*
3. c *occurs in* (t, σ),

there exists some type S in the derivation of $\emptyset \vdash e_0 \hookrightarrow t_0 : T$ such that $height(c) \leq height(S)$ and $size(c) \leq 5(2^{height(S)} - 1)$.

Proof Induction on the length of the reduction sequence. The base case proceeds by induction on the derivation of $\emptyset \vdash e_0 \hookrightarrow t_0 : T$. In cases [C-App1], [C-App2], [C-Deref2], [C-Assign1], and [C-Assign2], the new coercions are introduced by applications of $coerce(S, T)$ for types S and T in the derivation. In each such case, Lemma 3 guarantees that $height(c) \leq max(height(S), height(T))$, and by Lemma 2, $size(c) \leq 5(2^{height(max(height(S),height(T)))} - 1)$. The remaining cases are trivial inductions.

If the reduction sequence is non-empty, we have $t_0, \emptyset \longrightarrow^k t', \sigma' \longrightarrow t, \sigma$. If c occurs in t', σ' then by induction, there exists an S in the derivation of $\emptyset \vdash e_0 \hookrightarrow t_0 : T$ such that $height(c) \leq height(S)$ and $size(c) \leq 5(2^{height(S)} - 1)$. Otherwise, the only reduction rules that produce fresh coercions are [E-CBeta], [E-CDeref], [E-CAssign], all of which produce strictly smaller coercions, and [E-CCast]. In the latter case, the fresh coercion $c = c_1; c_2$ is synthesized from coercions c_1 and c_2 in the redex. By induction, there are types S_1 and S_2 in the derivation of $\emptyset \vdash e_0 \hookrightarrow t_0 : T$ such that $height(c_i) \leq height(S_i)$ and $size(c_i) \leq 5(2^{height(S_i)} - 1)$ for $i \in 1, 2$. Since $height(c) = max(height(c_1), height(c_2))$, let S' be the taller of the two types. Then $height(c) \leq height(S')$ and by Lemma 2, $size(c) \leq 5(2^{height(c)} - 1)$, so $size(c) \leq 5(2^{height(S')} - 1)$. □

We now bound the total cost of maintaining coercions in the space-efficient semantics. We define the size of a program state inductively as the sum of the sizes of its components. In order to construct a realistic measure of the store, we count only those cells that an idealized garbage collector would consider live by restricting the *size* function to the domain of reachable addresses.

$$
\begin{aligned}
&size(t, \sigma) = size(t) + size(\sigma|_{reachable(t)}) \\
&size(\sigma) = \textstyle\sum_{a \in dom(\sigma)}(1 + size(\sigma(a))) \\
&size(k) = size(a) = size(x) = 1 \\
&size(\lambda x{:}T.\ t) = 1 + size(T) + size(t) \\
&size(\mathtt{ref}\ t) = size(!t) = 1 + size(t) \\
&size(t_1 := t_2) = size(t_1\ t_2) = 1 + size(t_1) + size(t_2) \\
&size(\langle c \rangle\ t) = 1 + size(c) + size(t)
\end{aligned}
$$

To show that coercions occupy bounded space in the model, we compare the size of program states in reduction sequences to program states in an "oracle" semantics where coercions require no space. The oracular measure $size_{\mathrm{OR}}$ is defined similarly to *size*, but without a cost for maintaining coercions; that is, $size_{\mathrm{OR}}(c) = 0$. The following theorem then bounds the fraction of the program state occupied by coercions in the space-efficient semantics.

Theorem 5 (Space consumption) *If $\emptyset \vdash e \hookrightarrow t : T$ and $t, \emptyset \longrightarrow^* t', \sigma$, then there exists some type S in the derivation of $\emptyset \vdash e \hookrightarrow t : T$ such that $size(t', \sigma) \in O(2^{height(S)} \cdot size_{\mathrm{OR}}(t', \sigma))$.*

Proof During evaluation, the [E-CCast] rule prevents nesting of adjacent coercions in any term in the evaluation context, redex, or store. Thus the number of coercions in the program state is proportional to the size of the program state. By Theorem 4 the size of each coercion is in $O(2^{height(S)})$ for the largest S in the typing of e. □

8.2 Implementing tail recursion

Theorem 5 has the important consequence that coercions do not affect the control space consumption of tail-recursive programs. This important property is achieved by always combining adjacent coercions on the stack via the [E-CCAST] rule. This section sketches three implementation techniques for this rule.

Coercion-passing style This approach is similar to security-passing style [36], which adds an extra argument to every procedure representing its security context; in this case, the argument represents the result coercion. Thus the representation of a function type is

$$\lceil S \to T \rceil = \lceil S \rceil \times \mathit{Coercion} \to \lceil T \rceil$$

Tail calls coalesce but do not perform coercions, instead passing them along to the next function. Coercions are instead discharged when a value is returned to a context with a non-trivial coercion.

Trampoline A trampoline [18] is a well-known technique for implementing tail-recursive languages where tail calls are implemented by returning a thunk to a top-level loop. Tail-recursive functions with coercions return both a thunk and a coercion to the driver loop, which accumulates and coalesces returned coercions. The representation of a function type is then

$$\lceil S \to T \rceil = \lceil S \rceil \to (\lceil T \rceil + ((\mathbf{1} \to \lceil T \rceil) \times \mathit{Coercion}))$$

Continuation marks Continuation marks [9, 10] allow programs to annotate continuation frames with arbitrary data. When a marked frame performs a tail call, the subsequent frame can inherit and modify the destroyed frame's marks. Coercions on the stack are stored as marks and coalesced on tail calls. In this approach, the representation of a function type is direct:

$$\lceil S \to T \rceil = \lceil S \rceil \to \lceil T \rceil$$

since the presence of continuation marks makes the target language expressive enough to store coercions on the stack. However, the implementation of tail calls requires a protocol for coalescing coercions, so compilation is still required.

9 Related work

There is a large body of work combining static and dynamic typing. The simplest approach is to use reflection with the type `Dyn`, as in Amber [6]. Since case dispatch cannot be precisely type-checked with reflection alone, many languages provide statically-typed `typecase` on dynamically-typed values, including Simula-67 [2] and Modula-3 [7].

For dynamically-typed languages, *soft typing* systems provide type-like static analyses for optimization and early error reporting [37]. These systems may provide static type information but do not allow explicit type annotations, whereas enforcing documented program invariants (i.e., types) is a central feature of gradual typing.

Similarly, Henglein's theory of dynamic typing [22] provides a framework for static type optimizations but only in a purely dynamically-typed setting. We use Henglein's coercions instead for structuring the algebra of our cast representation. Our application is essentially

different: in the gradually-typed setting, coercions serve to enforce explicit type annotations, whereas in the dynamically-typed setting, coercions represent checks required by primitive operations.

None of these approaches facilitates *migration* between dynamically and statically-typed code, at best requiring hand-coded interfaces between them. The gradually-typed approach, exemplified by Sage [20] and $\lambda^{?}_{\rightarrow}$ [32], lowers the barrier for code migration by allowing mixture of expressions of type `Dyn` with more precisely-typed expressions. Our work improves gradual typing by eliminating the drastic effects on space efficiency subtly incurred by crossing the boundary between typing disciplines. Siek and Taha's subsequent work on gradual typing for objects [33] merges some casts at runtime, but they do not address tail recursion and so the remaining casts may still occupy unbounded space. In particular, our Space Consumption Theorem does not hold in their setting.

In earlier work in this area, [28] explored similar ideas in the context of dependent type systems. Their approach uses function proxies to support clean interoperation and hence migration between simply-typed code and dependently-typed code. Several other systems employ dynamic function proxies, including hybrid type checking [16], software contracts [15], and recent work on software migration by Tobin-Hochstadt and Felleisen [34, 35]. We believe our approach to coalescing redundant proxies could improve the efficiency of all of these systems.

Minamide and Garrigue [27] address a similar issue with unbounded proxies but in the context of specialization of polymorphic functions rather than general type conversions. Their approach does not handle tail calls correctly and only addresses runtime complexity. Our work presents a representation of casts as coercions in order to improve the memory guarantees of gradual typing. Nevertheless, our representation appears to improve the runtime efficiency similarly to Minamide and Garrigue's work. We intend to explore the runtime improvements of our approach as well.

10 Conclusion

We have presented a space-efficient implementation strategy for the gradually-typed λ-calculus. By ensuring that the presence of coercions does not asymptotically affect the space consumption of programs, we guarantee that programmers can gradually annotate programs with types without inadvertently introducing significant changes in resource consumption. Since the publication of our original conference results [23], other researchers have extended these ideas to support blame tracking [31] and to express normalized coercions as more elegant threesomes [30].

Most work on hybrid and gradual type systems to date has been developed in the context of rather idealized languages. The motivation for these idealized languages has been to enable researchers to focus on some of the essential and theoretical complexities of these type systems. However, some systems do not support side effects, and others, such as [19], assume the language is strongly-normalizing.

There remain some difficulties in scaling up these ideas to realistic languages with full-featured type systems that include features such as classes, subtyping, method overriding, recursive types, or polymorphism, particularly while preserving the desirable property of space efficiency.

Polymorphic types present some challenges for maintaining constant bounds on the size of coercions, in part because polymorphic types require representation of data structures with binders. The integration of polymorphism into various contract systems and hybrid/gradual type systems has been a topic of active research over the last several years.

Polymorphism was incorporated into dynamic contracts by Guha et al. [21]. More recently, gradual typing was extended to support polymorphism, using dynamic sealing to ensure that values of polymorphic type satisfy relational parametricity [1]. Neither of these results investigate space overheads.

Predicate types pose additional challenges for bounding the size of coercions, and may require additional assumptions, such as a bound on the number of predicates used in types. Combining polymorphism with predicate types may be tricky, and most research restricts predicates to base types. Some systems support predicates on non-base types (see for example, [3, 20, 38]), and this generalization seems necessary to support polymorphism, since predicates could be used to refine polymorphic arguments. In particular, the programming language SAGE [20] was an experiment at applying hybrid typing to a rich type system with first-class types. First-class types can then be used to express explicit polymorphism, where type parameters are passed explicitly, just like other parameters. Unfortunately, the lack of type inference in SAGE made type annotations and explicit type parameters somewhat burdensome. There was no evaluation of the space overheads in SAGE.

The integration of recursive types into hybrid/gradual type systems also deserves further study. Contract systems have long supported recursive contracts (for example, a list contract). SAGE's first-class types can express recursive types, but there are no results on the space overheads of the resulting coercions. It appears fairly plausible that the recursive types can be supported in a gradual type system in a space-efficient manner, although the details remain to be worked out.

In addition to the technical difficulties of scaling gradual and hybrid types to rich languages in a space-efficient manner, another key challenge for future work is to demonstrate the software engineering benefits of these techniques for code development and evolution.

References

1. Ahmed, A., Findler, R.B., Matthews, J., Wadler, P.: Blame for all. In: Workshop on Script to Program Evolution (2009)
2. Birtwhistle, G.M., et al.: Simula Begin. Chartwell-Bratt Ltd., Bromley (1979)
3. Blume, M., McAllester, D.: Sound and complete models of contracts. J. Funct. Program. **16**(4–5), 375–414 (2006). ISSN 0956-7968. doi:10.1007/s10990-011-9066-z
4. Bracha, G.: Pluggable type systems. In: Workshop on Revival of Dynamic Languages, October 2004
5. Brus, T., van Eekelen, M., van Leer, M., Plasmeijer, M.: Clean nnn a language for functional graph rewriting. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science, vol. 274, pp. 364–384. Springer, Berlin (1987)
6. Cardelli, L.: Amber. In: Spring School of the LITP on Combinators and Functional Programming Languages, pp. 21–47 (1986)
7. Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G.: Modula-3 report (revised). Technical Report 52, DEC SRC (1989)
8. Chambers, C.: The Cecil Language Specification and Rationale: Version 3.0. University of Washington (1998)
9. Clements, J.: Portable and high-level access to the stack with Continuation Marks. PhD thesis, Northeastern University (2005)
10. Clements, J., Felleisen, M.: A tail-recursive machine with stack inspection. ACM Trans. Program. Lang. Syst. **26**(6), 1029–1052 (2004)
11. de Oliveira, R.B.: The Boo programming language (2005)
12. ECMAScript Edition 4 group wiki. Ecma International (2007)
13. ECMAScript Language Specification. Ecma International, 5th edn. (2009). URL http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf
14. Findler, R.B., Blume, M.: Contracts as pairs of projections. In: Proceedings of the International Symposium on Functional and Logic Programming (FLOPS), pp. 226–241 (2006)
15. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 48–59, October 2002

16. Flanagan, C.: Hybrid type checking. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 245–256 (2006)
17. Flanagan, C., Freund, S.N., Tomb, A.: Hybrid types, invariants, and refinements for imperative objects. In: The International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD) (2006)
18. Ganz, S.E., Friedman, D.P., Wand, M.: Trampolined style. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 18–27 (1999)
19. Greenberg, M., Pierce, B., Weirich, S.: Contracts made manifest. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2010)
20. Gronski, J., Knowles, K., Tomb, A., Freund, S.N., Flanagan, C.: Sage: Hybrid checking for flexible specifications. In: Scheme and Functional Programming Workshop, September 2006
21. Guha, A., Matthews, J., Findler, R.B., Krishnamurthi, S.: Relationally-parametric polymorphic contracts. In: DLS, pp. 29–40 (2007)
22. Henglein, F.: Dynamic typing: Syntax and proof theory. Sci. Comput. Program. **22**(3), 197–230 (1994)
23. Herman, D., Tomb, A., Flanagan, C.: Space-efficient gradual typing. In: Trends in Functional Programming (TFP) (2007)
24. Knowles, K.W., Flanagan, C.: Hybrid type checking. ACM Trans. Program. Lang. Syst. **32**(2) (2010)
25. Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2007)
26. Meijer, E., Drayton, P.: Static typing where possible, dynamic typing when needed. In: Workshop on Revival of Dynamic Languages (2005)
27. Minamide, Y., Garrigue, J.: On the runtime complexity of type-directed unboxing. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 1–12 (1998)
28. Ou, X., Tan, G., Mandelbaum, Y., Walker, D.: Dynamic typing with dependent types. In: Proceedings of the IFIP International Conference on Theoretical Computer Science, pp. 437–450 (2004)
29. Pil, M.: First class file I/O. In: IFL '96: Selected Papers from the 8th International Workshop on Implementation of Functional Languages, pp. 233–246, London, UK. Springer, Berlin (1997). ISBN 3-540-63237-9
30. Siek, J., Wadler, P.: Exploring the design space of higher-order casts. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2010)
31. Siek, J., Garcia, R., Taha, W.: Exploring the design space of higher-order casts. In: Proceedings of the European Symposium on Programming Languages and Systems (ESOP), pp. 17–31. Springer, Berlin (2009)
32. Siek, J.G., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop, September 2006
33. Siek, J.G., Taha, W.: Gradual typing for objects. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Berlin, Germany, July 2007
34. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. In: Dynamic Languages Symposium, October 2006
35. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 395–406 (2008)
36. Wallach, D.S.: A New Approach to Mobile Code Security. PhD thesis, Princeton University, Department of Computer Science (1999)
37. Wright, A.K.: Practical Soft Typing. PhD thesis, Rice University (August 1998)
38. Xu, D.N., Peyton Jones, S.L., Claessen, K.: Static contract checking for Haskell. In: POPL, pp. 41–52 (2009)