SATySFI におけるドメイン固有型エラー診断

小林 亮太　佐藤 重幸　田浦 健次朗

EDSL はホスト言語に対する抽象化を提供し，事前のプログラミング知識がなくても利用できる点で有用である．しかし，EDSL の型エラーは通常ホスト言語で生成され，そのまま EDSL のユーザに提示されるため，型エラーによって抽象化が破壊されてしまうという問題がある．そこで本研究では，関数型組版言語 SATySFI において，[Serrano & Hage '19] に基づくドメイン固有型エラー診断を実現する言語拡張を提案する．この拡張によって，文書記述を目的とした SATySFI 利用者にとって分かり易いエラーメッセージを，文書記述用 EDSL 開発者が手軽に提供できるようになる．現状の実装においては，関数定義において満たされるべき型制約とその制約が満たされなかった場合に型エラーを分類するための制約を記述することで，間違え方に応じたエラーメッセージのカスタマイズを実現している．

1　はじめに

ドメイン固有言語（DSL）は，特定のドメインに特化したプログラミング言語であり，利用者に要求する前提知識を通常のプログラミング言語よりも抑えることができることから幅広く利用されてきた．DSL の中でも別の言語に埋め込まれる形で提供されるものは Embedded DSL (EDSL) と呼ばれ，TyXML [7] や SwiftUI [5]，Eigen [3] など採用例が多数存在する．EDSL は埋め込み対象の言語（ホスト言語）に対する抽象化を提供しているとも考えられ，開発にホスト言語の資産を活用しつつも EDSL 利用者にはホスト言語の経験を要求しないインターフェースを構築可能である．

しかしながら，EDSL の提供する抽象化は型エラーによって破壊されてしまうことが Hage によって指摘されている [9]．これは，EDSL における型エラーは通常ホスト言語で生成され，それが加工されることなく EDSL のユーザに提示されることが原因である．すなわち，型エラーは抽象化の外で生成されるため，その解読にはホスト言語の知識が必要となってしまうという問題である．これに対し Serrano らはドメイン固有の情報を型検査の各フェーズで注入し，エラーメッセージをカスタマイズする手法を提案しているが [12]，実世界の言語における言語拡張の実現方法やその評価については未だ研究余地があるほか，複数の束縛に跨ってエラー修正候補を探索できないといった制限がある．

そこで本研究は，SATySFI という言語を題材として，抽象化を崩さないドメイン固有型エラーを実現する言語拡張を提案する．中心となるのは型制約変数と呼ばれる型制約の計算を遅延する機構であり，単一の束縛対象に制限されず，SATySFI における EDSL 記述の特徴を活かした修正候補探索を実現する．

本研究の貢献は以下の通りである．

- ML 系言語に対するドメイン固有型エラー診断拡張を提案した．(第 5 章)
- SATySFI に提案拡張を実装し，予備実験を行った．(第 6 章)

Domain-specific Type Error Diagnosis in SATySFI.

Ryota Kobayashi, Shigeyuki Sato, and Kenjiro Taura, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

$$\frac{\alpha : \text{fresh} \quad \Gamma \vdash e_1 : \tau_1 \rightsquigarrow C_1 \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \vdash e_1\ e_2 : \alpha \rightsquigarrow C_1 \cap C_2 \cap \{\tau_1 = \tau_2 \to \alpha\}}\ \textsc{App}$$

$$\frac{\alpha : \text{fresh} \quad \Gamma, x : \alpha \vdash e : \tau \rightsquigarrow C}{\Gamma \vdash \lambda x.\ e : \alpha \to \tau \rightsquigarrow C}\ \textsc{Abs}$$

図 1 型付け規則

2 制約ベースの型検査と型推論

本章では，静的型付き言語において型検査や推論がどのように行われているのかを解説する．型検査器の実装にはいくつかのスタイルが存在するが，ここでは型制約を用いる手法を紹介する．この手法は GHC Haskell [14] や Swift [6]，Elm [2] のコンパイラでも利用されているものである．こうした型検査器での処理に沿って，制約の生成（2.1）と解消（2.2）に分けて説明する．

2.1 制約生成

このフェーズでは，抽象構文木をトラバースして各要素に対応する型制約を生成する．この際型判定は以下のように記述される．

$$\Gamma \vdash e : \tau \rightsquigarrow C$$

これは，制約群 C が成り立つ時，型環境 Γ において式 e が型 τ を持つことを意味する．

型付け規則の例を図 1 に示す．App 規則は，関数適用 $e_1\ e_2$ が型 α（α は新たに導入された型変数）を持つために，1) e_1 が制約群 C_1 を条件として型 τ_1 を持つこと，2) e_2 が制約群 C_2 を条件として型 τ_2 を持つこと，3) τ_1 と $\tau_2 \to \alpha$ が等しいことが必要であることを示す．また，Abs 規則は，ラムダ抽象 $\lambda x.\ e$ が型 $\alpha \to \tau$ を持つためには，x が型 α を持つと仮定したときに e が型 τ と評価できる必要があり，そこでの条件 C が引き継がれることを示す．このような過程を経て抽象構文木全体に対応する制約群が収集され，次の制約解消の段階へと渡される．

2.2 制約解消

前節の処理で収集した制約群を型に関する連立方程式とみなし，各型変数に当てはまる型を求める．等式制約のみからなる場合は，各制約について順に両辺の型の単一化を行えば良い．この計算によって，全ての型変数に対応する型が矛盾なく求まれば型検査に通過したとみなせる．一方，条件不足により定まるべき型変数の型を定めることができなかったり，制約解消中に矛盾が生じた場合は型エラーとなり，ユーザに対してエラーメッセージの形で報告される．

3 SATySFi

SATySFi [4] とは，TeX/ LaTeX の後継を目指す組版システム及びその上で使用される言語である．TeX/ LaTeX における「エラーメッセージがわかりにくい」という課題を解決することを目指しており，ユーザの入力に誤りがあった場合に，通常の静的型付き言語と同様誤りを含むプログラム上の位置と型レベルでのエラー内容が報告される．SATySFi には OCaml に類似した構文でコマンドや関数が定義できるプログラムレイヤーと，LaTeX に類似した構文で文書構造を記述できるテキストレイヤーの 2 つが存在し，一般的なユーザはほぼテキストレイヤー上での記述のみで文書作成を実現できることが特徴である．

SATySFi や LaTeX のような組版言語が汎用言語と異なる点は，利用者が必ずしもプログラミングに慣れていないという点である．こうした言語では利用の前提となる知識を可能な限り少なくすることが重要となる．例えば，SATySFi 上でクラスファイルやライブラリを利用する際には，それらで用いられるいくつかのコマンドの使い方さえ理解していれば良いというのが理想的である．こうした要求を実現するため，SATySFi 上でよく用いられるアプローチが次節で紹介する EDSL である．

3.1 EDSL

EDSL (Embedded Domain-Specific Language) とは，汎用言語に埋め込む形で提供される，特定のドメインに特化した言語である．この際埋め込み先の言語はホスト言語と呼ばれる．こうした言語は C 言語

や Python などの通常のプログラミング言語とは異なり，もともと意図された用途以外に汎用的に用いることは難しい．しかしながらその分必要な知識が制限されており，汎用プログラミング言語の経験が必須とならないことが利点としてあげられる．また，ホスト言語の視点ではライブラリであるため，既存のコード資源やエコシステムを活用でき，新たに独立した言語を作るよりも開発コストが小さい点でも優れている．

コード 1 は，SATYSFI 上で実現可能な EDSL の例である．この EDSL はパスの描画を目的としたもので，通過点の列としてパスを表現し，各点の座標を `--` で区切って記述する（6 行目）．ただし，パスの開始点には `start-path` の前置が必要である．なお，`0mm` などは SATYSFI 上で長さを示すリテラルで，`length` 型を持つ．

このような EDSL を SATYSFI 上で実現するためには，`start-path` や `--` を 1〜4 行目のような関数・演算子として定義すれば良い．`pre-path` は作成途中のパスを意味する SATYSFI の組み込み型である．`start-path` は初期座標を `length * length` として受け取り，初期座標のみを含む `pre-path` を返す関数である．`--` は `pre-path` と次に進む座標を受け取り，その座標をパスの末尾に追加した新しい `pre-path` を返す演算子である．上記の 4 行目の例の場合，`start-path (0mm, 0mm)` で初期パスを生成し，その後左結合の演算子 `--` で区切りながら座標を追加していくことでパスを作成している．パスの記述に必要なのは 1) 座標を `(0mm, 0mm)` のような形式で表現すること，2) `--` で区切りながら点を並べていくこと，3) 先頭に `start-path` をつけることの 3 点のみで，EDSL として必要知識の制限ができていることがわかる．

3.2 EDSL での型エラー診断

EDSL で問題となるのは，言語が提供する抽象化がエラーメッセージによって崩されてしまい得るということである [9]．ここでは，静的型付き言語をホスト言語とする EDSL を考える．まず，EDSL 自体はホスト言語上で作成されているため，当然その開発にはホスト言語の知識が必要となる．一方で，EDSL のユーザは EDSL のドメイン知識さえあればその言語を利用することができる．すなわち，EDSL は一種の抽象化のレイヤーと考えることができ，限られた知識のみで扱えるインターフェースとして機能する．しかしこの抽象化を破壊してしまうのがエラーメッセージである．通常，EDSL ではエラーメッセージはホスト言語によって生成され，特に加工されることなく EDSL のユーザに提示される．すなわち，EDSL が仮定するドメイン知識によらず，EDSL の内部実装に応じたエラーが出力されてしまう可能性がある．この解読には当然ホスト言語の知識が必要で，EDSL が実現すべき知識の制限や抽象化が壊れてしまうのである．

```
1 val start-path :
2   length * length -> pre-path
3 val (--) :
4   pre-path -> length * length -> pre-path
5
6 start-path (0mm, 0mm) -- (10mm, 0mm) --
      (10mm, 10mm)
```

コード 1 SATYSFI 上での EDSL の例

4 先行研究

前章の問題を解決するため，Serrano らは EDSL の作者が型エラーをカスタマイズできるアーキテクチャを提案している [12]．このアーキテクチャのもとでは，制約生成時に制約に対しコンテキストに応じたメッセージを付与することができ，その制約が型エラーの原因となった際に対応するメッセージを表示することで，エラーを補足することが可能となっている．また，ある制約が型エラーの原因となった場合，予め指定しておいた別の制約（代替制約）に取り替えて再度型検査を行い，もしそれによって型検査に通過するようになれば，その取り替えに対応する修正候補を提示することも可能としている．

例えば，コード 1 の EDSL の場合，想定されるよくあるミスとして `start-path` の書き忘れがある．すなわち，パス描画の際は最初の座標の前に `start-path` を置く必要があるが，それを忘れて `(0mm, 0mm) --`

(10mm, 0mm) のように書いてしまうというミスである．この場合，-- 演算子の左辺が pre-path ではなく length * length になるため，SATySFi のコンパイラは以下のようなエラーを出力する（本質的な部分のみを抜粋）．

```
! [Type Error] at ...
    this expression has type
      length * length,
    but is expected of type
      pre-path.
```

静的型付き言語のエラーメッセージとしては必要十分だが，SATySFi でユーザとして想定するプログラミング初学者にとっては，このメッセージの解読は難しいだろう．今回の場合は先頭に start-path 関数をつけることがエラー解決の正しい方法だが，それをこのエラーメッセージだけから読み取ることは困難である．こうした状況でユーザを的確に誘導するためには，コンパイラが以下のように具体的な修正方法を提示することが望ましい．

```
You might want to enclose the first point with
    'start-path'
```

このようなエラーメッセージを実現するためには，「--演算子の左辺に length * length が渡され型エラーとなった」ことを検知し，メッセージを差し替える必要がある．Serrano らの手法では特定の関数適用から生じた型制約のみにメッセージや代替制約を指定することが可能であり，今回の場合は以下のような指示を注入することでエラーメッセージのカスタマイズを実現できる．

1. -- 演算子の左辺の型に関する条件を制約としてくくりだす．具体的には，-- 演算子の型を $\alpha \to \text{length}*\text{length} \to \text{pre-path}$ とした上で，制約 $\alpha = \text{pre-path}$ を導入する．
2. $\alpha = \text{pre-path}$ の代替制約として，$\alpha = \text{length}*\text{length}$ を登録する．前者の制約が満たされず型エラーになった場合，後者の制約に置き換えた上で型検査が再実行される．もしこの置き換えによって型エラーが解消されるならば，上記のカスタマイズされたメッセージを表示する．

```
1 let f p = p -- (0mm, 0mm) in
2 (f (10mm, 0mm), f (0mm, 10mm))
```

コード 2　型エラーとなる SATySFi プログラム例

こうした型エラー発生時の修正候補探索は，束縛グループ（binding group）単位で行われる．すなわち，型エラーが発見された場合に，その原因の探索対象は現在検査中の束縛グループに限定され，それ以前に型検査した束縛グループは正しいものとして扱うということである．したがって，この手法では複数の束縛に跨ってエラー修正を検討することができない．例えば，コード 2 のようなプログラムにおいて，1 行目の関数 f の引数 p の型は p -- (0mm, 0mm) によって pre-path と推論され，2 行目の f (10mm, 0mm) で矛盾が生じて型エラーとなる．この際，Serrano らの手法では束縛単位で型を確定させているため，1 行目の f の定義にエラー原因が存在する可能性を考慮して修正候補を提示することができないという制限がある．

先行研究の対象とする Haskell では，トップレベルの束縛には型アノテーションをつけることが一般的であり，以前に検査した束縛の型は正しいものとして扱っても問題ないことも多い．一方，OCaml や本研究が対象とする SATySFi では明記しない場合も多く，ユーザの意図しない型になりやすい．したがって，修正候補の探索範囲も広い方が望ましいと考えられる．

5　提案手法

本研究では，Serrano らの手法を拡張し，複数の let 束縛に跨って修正候補を探索できる型エラー診断を SATySFi に実装する．すなわち，制約を各束縛単位ではなく，より広い範囲でまとめて解消することで，探索範囲を広げることを目指す．

ここで問題となるのは，let 多相との兼ね合いである．SATySFi においては，OCaml 等と同様 let で定義された変数は多相型を持ち，型に含まれる型変数の一部が量化されている．プログラム中でその変数が利用される場合，量化された型変数はフレッシュ

な型変数によって置き換えられる（instantiate）．ここで，let による束縛対象の式に制約が紐づけられていた場合，その制約は変数の利用箇所でも検証されなければならない．例えばコード 2 では，`--` 演算子に紐づけられた制約 $\alpha = \text{pre-path}$ を，`f` の各利用箇所でも検証する必要がある．

これに対する単純な解決策は，変数の利用のたびに制約をコピーすることである（図 2）．この際，制約内に含まれる型変数は，型本体での型変数の置換に合わせて置き換える．図 2 の例の場合，`f` は $\alpha \to \text{pre-path}$ という型を持ち（α は `p` の型），$\alpha = \text{pre-path}$ という `--` 演算子から引き継がれた制約が束縛対象に紐づいている．そのため，`f` の利用箇所ではこの制約がコピーされ，`f` 本体の型に同期して型変数が置き換えられる．しかし，このように制約をコピーしてしまうと，代替制約の選択を伝搬させることが難しい．図 2 で言えば，`--` 演算子に紐づけられた制約 $\alpha = \text{pre-path}$ の代わりに $\alpha = \text{length} * \text{length}$ を選択した場合，$\beta = \text{pre-path}$, $\gamma = \text{pre-path}$ でも制約を置き換える必要があるが，単純な制約のコピーではこうした制約の依存関係が失われてしまうためである．

この問題の解決のため，本研究では型制約変数 ϕ を導入する（図 3）．この変数は後に具体的な型制約が代入されるもので，型変数の制約版と考えられる．ここでは，依存関係のある制約には同じ型制約変数が割り当てられるとしているため，これらの制約を連動して変化させることが可能となる．また，変数を instantiate した際の型変数の置換 θ を別に記録しておくことで，各箇所の制約を後で（制約解消フェーズで）計算できるようにしている．プログラム中の各項には型制約変数と置換の組である制約参照 $r = (\theta, \phi)$ のリストが紐づけられており，制約生成フェーズで再帰的に収集される．各型制約変数にどのような制約が代入されうるかは制約選択マップ Σ として別に管理されており，これもプログラム中の各項に紐づけられ収集される．制約生成フェーズではこうした制約参照のリスト R と制約選択マップ Σ を生成し，制約解消フェーズにおいてこれらから具体的な制約を計算した上で解消を試みることになる．以降 5.1, 5.2 で具体的な処理の流れを解説する．

5.1 制約生成

ここでは簡単のため，SATySFi のサブセットである以下の言語を考える．これは単純型付きラムダ計算に let 多相を加えたものである．

$$e ::= x \mid e_1\ e_2 \mid \lambda x.\ e \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$$

制約生成は図 4 の規則に基づいて行う．型判定は $\Gamma \vdash e : \tau \rightsquigarrow (R, \Sigma)$ と書き,「式 e は型環境 Γ の下で型 τ を持ち，制約参照リスト R と選択マップ Σ を生成する」と読むこととする．

また，型環境も拡張し $\Gamma ::= \left\{\overline{x : (R, S) \Rightarrow \sigma}\right\}$ とする．これは，型本体の σ に加えて，その変数がプログラム中で使用された場合にコピーする制約参照 R と新たに生じる選択 S を登録するものである．R には let 束縛の束縛対象から引き継がれた制約参照が入り，S には EDSL の作者によって型エラー診断用に明示された制約が入る．なお，ここでは型エラー診断用の制約記述場所としてモジュールシグネチャを想定しており（後述），通常の let 束縛に対し型エラー診断用の制約を記述することはできないとしているため，R, S のうち少なくとも片方は空となる．

上記の型環境に記録された変数は，Var 規則によって取り出される．型本体 σ は置換 θ によって instantiate され，取り出された項には，R や S から計算された制約参照や制約選択マップが紐づけられる．R, S のいずれかは空であるため，それらを分けて考えると，

> R が空でない場合（let 束縛により導入）：$R = (\theta_0, \phi_0)$ に対して置換 θ を適用した $\theta R = (\theta \circ \theta_0, \phi_0)$ を項に紐づける．
>
> S が空でない場合（モジュールシグネチャにより導入）：$|S|$ 個の新たな型制約変数を生成し，それぞれに値域 S_i を対応づけて選択マップ Σ を生成する．また，その型制約変数と置換 θ の組を制約参照として項に紐づける．

ここで重要となるのは型制約変数を変数使用のたびに生成していることである．これによって，各変数の使用位置で独立して修正候補を試すことができるようになる．

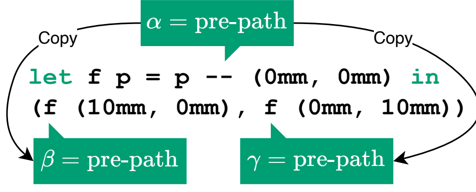

図 2　制約のコピーによる伝搬

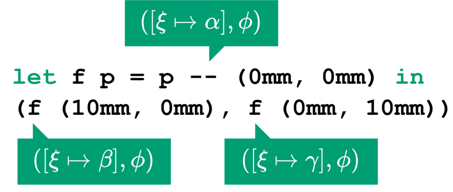

図 3　型制約変数による制約の伝播

$\boxed{\Gamma \vdash e : \tau \rightsquigarrow (R, \Sigma)}$

$$\frac{x : (R, S) \Rightarrow \sigma \in \Gamma \qquad \tau = \theta\sigma \qquad \overline{\phi_i} : \text{fresh}}{\Gamma \vdash x : \tau \rightsquigarrow (\theta R \mathbin{+\!\!+} [\overline{(\theta, \phi_i)}], \{\overline{\phi_i : S_i}\})}\ \textsc{Var}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \rightsquigarrow (R_1, \Sigma_1) \qquad \Gamma \vdash e_2 : \tau_1 \rightsquigarrow (R_2, \Sigma_2)}{\Gamma \vdash e_1\ e_2 : \tau_2 \rightsquigarrow (R_1 \mathbin{+\!\!+} R_2, \Sigma_1 \cap \Sigma_2)}\ \textsc{App}$$

$$\frac{\Gamma, x : ([], \emptyset) \Rightarrow \alpha \vdash e : \tau \rightsquigarrow (R, \Sigma) \qquad \alpha : \text{fresh}}{\Gamma \vdash \lambda x.\ e : \alpha \to \tau \rightsquigarrow (R, \Sigma)}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow (R_1, \Sigma_1) \qquad \sigma = \textsc{Gen}(\Gamma, \tau_1) \qquad \Gamma, x : (R_1, \emptyset) \Rightarrow \sigma \vdash e_2 : \tau_2 \rightsquigarrow (R_2, \Sigma_2)}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2 \rightsquigarrow (R_1 \mathbin{+\!\!+} R_2, \Sigma_1 \cap \Sigma_2)}\ \textsc{LetIn}$$

図 4　型付け規則

let 束縛は LetIn 規則により扱う．束縛対象 e_1 の型 τ_1 や紐づけられた制約参照 R_1 を元に e_2 の型検査を行うというのが基本的な流れである．e_2 内で x が出現した場合，制約の成立は各箇所で確認する必要があるが，各型制約変数への代入は元の式での判断に依存している．したがって，型環境中での x では R が束縛対象から引き継がれ，S は空になる．let 式全体では e_1 と e_2 に紐づく制約参照や選択マップが連結され，紐づけられる．

App, Abs 規則でも，制約参照や選択マップが再帰的に回収され，次の制約解消フェーズに送られる．

5.2　制約解消

制約解消は制約グループ単位で行う．ここでのグループとは，型エラーが発見された場合に修正候補を探索する範囲のことである．依存モジュールも含むプログラム全体の修正候補を試すと計算量があまりに増大してしまうことに加え，ユーザがあるライブラリの関数を利用する場合，そのライブラリ自体に誤りがあるとして修正候補を提示するのはあまり有用ではない．そのため，制約を適当な単位で分割し，その範囲内の代替制約のみを検討することにする．

ここではそうした制約グループの境界として，モジュールとコマンド引数の 2 種類を採用する．モジュールによる境界は，ライブラリ実装とその利用者を分割し，利用者側ではライブラリ実装自体の誤りは考慮しないことを示す．また，型エラー診断用の制約定義はモジュールのシグネチャ内で提供されることからも，モジュールを型エラー診断の境界とすることは妥当であると考えられる．一方コマンド引数による境界は，SATySFi 上での EDSL 実装のコンベンションに基づくものである．SATySFi 上で文書を作成する場合は通常テキストレイヤーを使用するが，その中で EDSL を記述する場合はコード 3 のようにコ

```
1 +p{
2   Hello, world.
3   \edsl(
4     start-path (0mm, 0mm) -- (10mm, 0mm)
5   );
6   Another text.
7 }
```

コード 3　SATySFi 上での EDSL 使用例

マンドの引数として与えられることが一般的である．すなわち，コマンドの引数が EDSL の意味的な境界になっており，エラー修正候補の探索範囲として妥当と考えられる．

制約解消のアルゴリズムを図 5 に示す．大まかに言えば，各制約グループについて以下の手順を踏むことでエラーメッセージを決定する．

1. 正常系（本来の制約）を選択し，それらの制約が解消できるかどうかを調べる（左側 2〜4 行目）．解消できれば型検査に通過したとみなす．
2. 失敗した場合，すなわち型エラーが存在する場合は，代替の制約を試す（6〜8 行目）．この際対象とする選択のうち，1 つだけを変えたもの，2 つだけを変えたもの，という順に試していく．
3. もし制約の取り替えによって型エラーが解消された場合は，対応するメッセージを出力する．
4. いずれの制約の組み合わせによっても型エラーが解消できなかった場合は，標準のエラーメッセージを出力する（9 行目）．

アルゴリズム中の各関数の意味は以下の通りである．

SelectDefaults: 選択マップ Σ から，正常系の制約を選択したものを返す．

Select: 選択マップ Σ から，n 個の制約について代替を選択し，それ以外は正常系を選択する．また，選択した代替制約に紐づけられたメッセージも返却する．通常複数パターンが存在するため，返り値はリストになる．

Instantiate: 制約参照 R と選択から，実際の制約を生成する．

Solve: 制約解消を行う．

上記のアルゴリズムにより，対象とする範囲内で最も少ない箇所を変更して型エラーを解消できるパターンを探索することができる．

6　予備実験

本章では，第 5 章で提案した型システムを SATySFi 処理系を拡張する形で実装し，その動作と型検査にかかる時間の変化を検証する．SATySFi には現行版（v0.0.x）と開発版（v0.1.x）が存在するが，本研究では開発版を対象とする．これらは構文やパッケージシステム，レコード多相のあり方等に違いがあるが，コマンド定義や文書記述の方法に大きな違いはなく，提案手法も両バージョンに適用可能なものである．

実行時間の計測にあたっては，実際の組版を行わず型検査のみを行う `--type-check-only` というオプションを利用した．また，SATySFi 実装中に OCaml の `Unix.gettimeofday` による時刻計測を埋め込むことで，実行時間の大半を占める 1) initialize（設定ファイルの読み込みやプリミティブのセットアップ）2) parsing dependent packages（依存パッケージのパース）3) parsing this package and make dependency graph（自身のパッケージのパースとパッケージ間の依存関係グラフの作成）4) type checking（型検査）という各フェーズの所要時間を計測した．なお，所要時間は 5 回の実行の平均を取っている．

ベンチマークとして使用した文書ファイルは SATySFiBook [13] である．この文書は A4 161 ページからなる SATySFi 自体の解説書であり，SATySFi で記述された文書としては特に長大なものである．オリジナルの SATySFiBook は現行版の SATySFi で記述されていたため，開発版に移植した上で実験に使用した．また，実験環境はいずれも 16 インチ MacBook Pro (2021, 16GB RAM) である．

6.1　実験結果

ここでは，型エラー診断用の情報を含まない SATySFiBook を型検査し，本研究における拡張によって生じるオーバーヘッドを確認した．結果は図 6 の通りである．まず，4 フェーズの実行の合計時間を比較すると，オリジナルの実装では 0.1676 秒，拡張したものは 0.1873 秒となり，約 1.1 倍の増加と

```
1:  procedure TrySolving(R, Σ)
2:      selected ← SelectDefaults(Σ)
3:      C ← Instantiate(R, selected)
4:      res ← Solve(C)
5:      if res = fail then
6:          for i ← 1, n do
7:              TryAlternatives(R, Σ, i)
8:          end for
9:          ShowDefaultMessages( )
10:     end if
11: end procedure
```

```
1:  procedure TryAlternatives(R, Σ, n)
2:      patterns ← Select(Σ, n)
3:      for (selected, messsages) ← patterns do
4:          C ← Instantiate(R, selected)
5:          res ← Solve(C)
6:          if res = success then
7:              Print(messages)
8:              Exit( )
9:          end if
10:     end for
11: end procedure
```

図 5　制約解消アルゴリズム

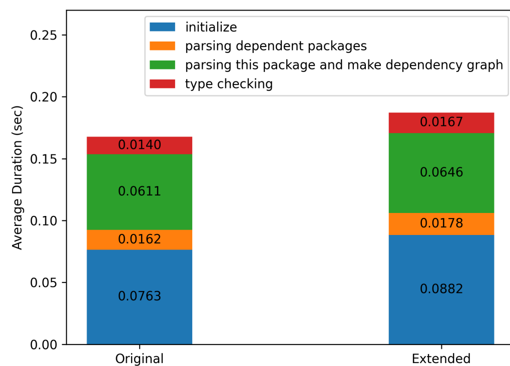

図 6　SATySFI の `--type-check-only` オプション下の実行時間の比較

なっている．また，型検査にかかる時間のみを比較すると約 1.2 倍の増加となっている．これらはいずれも小さく，元々コマンド実行にかかる時間が 0.2 秒以内であることを踏まえても，ユーザに対する悪影響は小さい．特に，型検査だけでなく実際の組版を行う場合はこの 10 倍以上の時間がかかることが一般的であり，ユーザに知覚できる差はないと考えられる．

また，プログラムに型エラー診断用の制約を加えた場合，型検査が正常に完了すること，その制約に反する記述をすると型エラーになること，その間違い方が事前に代替制約として想定したものであれば対応するメッセージが表示されることを確認した．

7　関連研究

既存言語における型エラーカスタマイズの実例としては，Haskell の Custom compile-time errors [1] が挙げられる．これは，型クラスの対応する実装が存在しない場合に，カスタマイズされたエラーメッセージを出力するものである．型クラス実装（instance）の構文を使用するため，どの型に対する実装を要求していたのかに応じてエラーメッセージを変化させることができ，Serrano らの手法における代替制約による修正候補提案に相当するエラーカスタマイズが実現できる．また，型族を使用して実装されているため，型エラー診断専用の構文が導入されていないことも特徴である．一方この手法では型エラーカスタマイズは型クラスに限定されており，通常の関数等には対応できない．

型エラー診断は様々なアプローチで研究されているが，その中でも特に多くなっているのが型エラーの原因の探索を改善するものである．例えば eCFT [8] は，型エラーが発生した場合に考え得るすべての修正候補を効率的に列挙可能なフレームワークである． eCFT では variational typing によって，プログラム中の各箇所を変化させた際に全体の型付けがどのように変化するかが計算される．さらに principal typing によって以前の計算結果を再利用できるようにすることで，型エラーが発生した際にプログラムをどのように変化させれば型エラーが修正できるかを

高速に列挙することを可能としている．一方，eCFTにおける修正候補はあくまでも型レベルであり，コンテキストに応じてエラーメッセージを指定できるものではない．本研究では，修正候補として提示する箇所を EDSL の作者が明示的に指定するとすることで，よりユーザにとって望ましい修正候補の提案を目指している．

Mycroft [11] は，型エラーの原因を制約レベルで絞り込むフレームワークである．型エラーを生じた制約集合から順に制約を除いていき，繰り返し制約解消を行うことで，矛盾を生じる最小の制約集合を得るものである．本研究の提案手法はグループ化した範囲の制約全てについて修正候補を検証するが，Mycroft のようなフレームワークで対象となる制約を最小化することで性能が改善する可能性があると考えている．

Hage と Heeren [10] は，制約の並べ替えにより，制約解消時に矛盾が発見される制約を制御するフレームワークを開発した．制約並べかえ戦略を切り替えることで各種型推論アルゴリズムをエミュレート可能で，コンパイラ実装時に特定の戦略をハードコードする必要がないことも利点である．本研究の提案手法では，制約の修正候補検証における優先順位はプログラムの構造から静的に定まるが，このフレームワークのようにカスタマイズ可能とすることも検討に値する．

[10] では let 多相における generalization, instantiation そのものを制約として扱うことで，制約をコピーせずに伝搬させる手法を採用していることも注目すべき点である．なお，本研究の提案手法では SATySFi の既存の貪欲な単一化による型推論アルゴリズムを変更せず，型エラー診断用に明示的に導入した型制約のみを制約解消フェーズにおいて解くとしているため，この手法が直接採用できるものではない．また，型エラーが発見された際に，その型エラーに関係のある制約のみを代替制約によって解き直すといった最適化が提案手法の今後の発展として考えられるが，こうした処理はこの手法では難しいと考えられ，型制約変数を用いた本研究の手法の利点となっている．

8 まとめと今後の課題

本稿では，組版言語 SATySFi において EDSL 向けの型エラー診断を実現する型システムを提案した．型エラーが EDSL の抽象化を破壊するという問題は特定の言語に依存しない普遍的な問題だが，利用者が必ずしもプログラミングに慣れていない SATySFi では特に問題となり，その解決にはコンパイラと EDSL 作者双方の貢献が欠かせない．提案手法は EDSL のユーザにとってわかりやすいエラーメッセージを，EDSL の作者が比較的小さい労力で実現することを可能にするものである．その設計に当たっては，Serrano らの手法を複数の let 束縛に跨ってエラー修正候補を探索できるよう拡張することで，ユーザにより多くの修正候補を提案することが可能となった．今後は第 7 章で紹介した関連研究を参考に，型エラーに関連する制約のみを取り出し修正候補探索を効率化させたり，複数の修正候補の優先度を柔軟に変化させることのできる仕組みを導入するといった方向が考えられる．

謝辞 SATySFi 処理系に関して有益な技術的コメントを頂いた諏訪敬之氏に感謝する．

参 考 文 献

[1] : 6.4.19. Custom Compile-Time Errors — Glasgow Haskell Compiler 9.7.20221225 User's Guide, https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/type_errors.html.
[2] : Compiler/Compiler/Src/Type/Solve.Hs at Master · Elm/Compiler, https://github.com/elm/compiler/blob/master/compiler/src/Type/Solve.hs.
[3] : Eigen, http://eigen.tuxfamily.org.
[4] : Gfngfn/SATySFi: A Statically-Typed, Functional Typesetting System, https://github.com/gfngfn/SATySFi.
[5] : SwiftUI, https://developer.apple.com/documentation/swiftui.
[6] : Type Checker Design and Implementation — Swift 2.2 Documentation, https://apple-swift.readthedocs.io/en/latest/TypeChecker.html.
[7] : TyXML, https://ocsigen.org/tyxml/.
[8] Chen, S. and Wu, B.: Efficient Counter-Factual Type Error Debugging, *Science of Computer Programming*, Vol. 200(2020), pp. 102544.
[9] Hage, J.: Solved and Open Problems in Type Error Diagnosis, *STAF Workshops*, (2020), pp. 62–74.

[10] Hage, J. and Heeren, B.: Strategies for Solving Constraints in Type and Effect Systems, *Electronic Notes in Theoretical Computer Science*, Vol. 236(2009), pp. 163–183.

[11] Loncaric, C., Chandra, S., Schlesinger, C., and Sridharan, M.: A Practical Framework for Type Inference Error Explanation, *ACM SIGPLAN Notices*, Vol. 51, No. 10(2016), pp. 781–799.

[12] Serrano, A. and Hage, J.: A Compiler Architecture for Domain-Specific Type Error Diagnosis, *Open Computer Science*, Vol. 9, No. 1(2019), pp. 33–51.

[13] Suwa, T.: The SATySFiBook, December 2018.

[14] Vytiniotis, D., Jones, S. P., Schrijvers, T., and Sulzmann, M.: OutsideIn(X) Modular Type Inference with Local Assumptions, *Journal of Functional Programming*, Vol. 21, No. 4-5(2011), pp. 333–412.