



پردیس علوم
دانشکده ریاضی، آمار و علوم کامپیوتر

بهبود واریسی مدل با استفاده از نظریه تعبیر مجرد

نگارنده

پویا پرتو

استاد راهنمای اول: دکتر مجید علی زاده
استاد راهنمای دوم: دکتر مجتبی مجتهدی

پایان نامه برای دریافت درجه کارشناسی ارشد
در رشته علوم کامپیوتر

تاریخ دفاع

چکیده

تقديم به

تقديم به

سپاسگزاری

سپاسگزاری

پیشگفتار

فهرست مطالب

۱	مقدمه	۱
۱	۱.۱ برخی از روش‌های درستی یابی برنامه‌ها	۱
۳	۲.۱ برخی مفاهیم اولیه	۳
۳	۱.۲.۱ نظریه تعبیر مجرد	۳
۳	۲.۲.۱ روش واریسی مدل	۳
۴	۲ صورتی‌گری جدید برای روش واریسی مدل	۴
۴	۱.۲ نحو زبان مورد بررسی	۴
۶	۲.۲ معنانشناسی زبان مورد بررسی	۶
۶	۱.۲.۲ برچسب‌ها	۶
۷	۲.۲.۲ رد پیشوندی	۷
۷	۳.۲.۲ تعریف صورتی معنانشناسی رد پیشوندی	۷
۱۰	۳.۲ ویژگی‌های معنایی برنامه‌ها	۱۰
۱۰	۱.۳.۲ ویژگی‌های معنایی	۱۰
۱۱	۲.۳.۲ عبارات منظم	۱۱
۱۱	۳.۳.۲ زبان عبارات منظم	۱۱
۱۳	۴.۳.۲ معنانشناسی عبارات منظم	۱۳
۱۵	۳ واریسی مدل منظم	۱۵
۱۶	۴ واریسی مدل ساختارمند	۱۶
۱۷	۵ نتیجه‌گیری	۱۷

فصل ۱

مقدمه

با توجه به پیشرفت روز افزون علوم کامپیوتر و ورود کاربردهای آن به زندگی روزمره، پیشرفت در روش‌های ساخت و نگهداری برنامه‌ها نیازی آشکار به نظر می‌رسد. یکی از مسائل مهم در این زمینه بررسی صحت کارکرد برنامه‌های نوشته‌شده است. عدم صحت کارکرد برنامه‌های نوشته‌شده بسته به حساسیت کاری که یک برنامه انجام می‌دهد می‌تواند تبعات مختلفی داشته‌باشد. پرتاب ناموفق آریان ۵ [۱۶]، از مدار خارج شدن مدارگرد مریخ [۲] و تصادف هلیکوپتر چینوک [۱] چند نمونه از تبعات بزرگ این قضیه در گذشته بوده‌اند. برای کشف صحت کارکرد برنامه‌های کامپیوتری روش‌های متفاوتی ابداع شده‌اند که در ادامه به طور مختصر از آن‌ها یاد می‌کنیم اما پیش از آن به یک خاصیت مشترک همه‌ی این روش‌ها می‌پردازیم که ناکامل بودن است. منظور از ناکامل بودن این است که با استفاده از هیچ یک از روش‌هایی که داریم نمی‌توانیم هر خاصیتی را برای هر برنامه‌ای بررسی کنیم. به عبارت دیگر استفاده از هر روشی، محدودیت‌هایی دارد. و البته قضیه رایس [۱۹] به ما این تضمین را داده که روش کاملی اصلاً وجود ندارد. قضیه رایس به طور غیر رسمی بیان می‌کند که مسأله‌ی بررسی هر خاصیت غیر بدیهی، برای همه‌ی برنامه‌ها تصمیم ناپذیر است. این دلیلی بر این شده که روش‌های مختلفی برای این کار درست شوند که هر کدام می‌توانند حالت‌های خاصی از مسأله را حل بکنند.

۱.۱ برخی از روش‌های درستی یابی برنامه‌ها

یک دسته‌بندی برای این روش‌ها تقسیم آن‌ها به دو دسته‌ی پویا و ایستا است. روش‌های پویا روش‌هایی هستند که در آن‌ها تست برنامه با اجرای برنامه همراه است و روش‌های ایستا بدون اجرای برنامه‌ها انجام می‌شوند. روش‌های پویا معمولاً با اجرای حالات محدودی از برنامه، تصمیم می‌گیرند که برنامه‌ای که نوشته‌ای ایم، انتظاراتمان را برآورده می‌کند یا خیر. اگر این روش بتواند تشخیص

دهد برنامه‌ای درست کار نمی‌کند می‌توانیم با اطمینان نتیجه بگیریم که برنامه غلط نوشته شده، اما اگر برنامه‌ای از تست‌های ساخته شده با این روش‌ها با موفقیت عبور کند، نمی‌توان اطمینان حاصل کرد که برنامه درست کار می‌کند، زیرا ممکن است حالتی مشکل‌زا از اجرای برنامه وجود داشته باشد که در تست‌ها نیامده باشد. در کتاب [۱۷] به توضیح این روش‌ها پرداخته شده. این دسته از روش‌ها از موضوع اصلی کار ما دور هستند. روش‌های ایستا معمولاً روش‌هایی هستند که از نظریه‌های مختلف در منطق ریاضی به عنوان ابزار بهره می‌برند تا بدون اجرای خود برنامه‌ها در مورد صحت اجرای آن‌ها نتیجه‌گیری کنند. به همین دلیل به بخشی مهم و بزرگی از این دستورات که از منطق استفاده می‌کنند روش‌های صوری هم گفته می‌شود. از معروف‌ترین این روش‌ها واریسی‌گر مدل، روش‌های استنتاجی و استفاده از نظریه تعبیر مجرد است. در روش واریسی مدل، یک مدل صوری متناهی از برنامه‌ی موردبررسی می‌سازیم که همه‌ی حالات اجرای برنامه با آن قابل‌توصیف است، سپس با استفاده از یک زبان صوری که بتواند در مورد مدل‌هایمان صحبت کند، ویژگی‌های مورد بررسیمان را بیان می‌کنیم و در نهایت صحت ویژگی‌های بیان‌شده را بررسی می‌کنیم. مقاله [۴] شروع این روش‌ها بوده که این کار را با استفاده از نوعی مدل کریپکی [۱۵] و نوعی منطق زمانی به نام منطق زمانی خطی [۴] انجام داده که روشی است با دقت و البته هزینه‌ی محاسباتی بسیار بالا. [۱۲] یک منبع بسیار مقدماتی در این زمینه و کتاب [۵] یک مرجع سنتی در این زمینه است. در روش‌های استنتاجی که شاید بتوان یکی از ابتدایی‌ترین آن‌ها را استفاده از منطق هور [۱۱] دانست، درستی کارکرد برنامه‌هایمان را با ارائه‌ی یک درخت اثبات در یک دستگاه استنتاجی که متناسب با زبان برنامه‌هایمان ساخته شده، نشان می‌دهیم. در این روش هم اگر بتوانیم درستی یک برنامه را اثبات کنیم، دیگر به‌طور تئوری، خیالی آسوده از درستی برنامه خواهیم داشت اما ساختن درخت اثبات در یک نظریه برهان می‌تواند چالش برانگیز باشد چون این یک مسئله‌ی NP-Hard است. در [۱۲] به منطق هور به‌طور مقدماتی پرداخته شده. همین‌طور کتاب [۱۸] نیز به پیاده‌سازی منطق هور در زبان coq پرداخته. coq نیز یک اثبات‌یار است که بر اساس یک نظریه نوع وابسته کار می‌کند. برای اطلاعات بیشتر در مورد چگونگی طرز کار این اثبات‌یار و تئوری بنیادین آن می‌شود به کتاب [۳] مراجعه کرد. تئوری مورد شرح در [۱۰] نیز می‌تواند در این مسیر به کار گرفته شود. نظریه تعبیر مجرد [۸] نیز یک نظریه ریاضیاتی است که در این بحث می‌توان گفت، به‌نوعی سعی می‌کند از روی معناشناسی یک برنامه‌ی کامپیوتری [۲۱]، یک تقریب بسازد. منظور از تقریب، یک دستگاه کوچک‌تر از معناشناسی اصلی است که رفتارش زیرمجموعه‌ی رفتارهای دستگاه اصلی است. سعی بر این است که دستگاه جدیدی که می‌سازیم به لحاظ محاسباتی هم ساده‌تر از معناشناسی اصلی کار کند تا بتوانیم خواص آن را راحت‌تر بررسی کنیم. در این صورت هر نتیجه‌ای که در مورد خواص جدید بگیریم را می‌توانیم در مورد خود برنامه هم بیان کنیم اما می‌دانیم که در این صورت هم به همه‌ی حقایق دست پیدا نکرده ایم. در مورد این نظریه نیز به‌تازگی کتاب [۷] منتشر شده که حاصل نزدیک به ۵ دهه کار مبدع این نظریه، پاتریک کوزو، است. همین‌طور [۱۳] نیز در مورد پیاده‌سازی این نظریه بحث کرده.

۲.۱ برخی مفاهیم اولیه

در این قسمت سعی شده خواننده با مفاهیم مقدماتی کار آشنا شود. بعضی از مفاهیم ممکن است مستقیماً در ادامه استفاده شده باشند و بعضی دیگر ممکن است مستقیماً در ادامه وارد بحث نشده باشند اما آشنایی با آنها برای درک بهتر واجب است.

۱.۲.۱ نظریه تعبیر مجرد

۲.۲.۱ روش واریسی مدل

فصل ۲

صوری‌گری جدید برای روش واریسی مدل

محوریت کار ما قرار است [۶] باشد که در آن سعی شده روش واریسی مدل با کمک نظریه تعبیر مجرد، بهبود داده شود. در [۴] روشی که معرفی شده در واقع ویژگی برنامه‌ها را به کمک منطق‌های LTL یا CTL بیان می‌کند. خود برنامه‌ها هم با کمک معنانشناسی این منطق‌ها که نوع خاصی از مدل‌های کرپسکی به اسم سیستم‌گذار هستند توصیف می‌شوند. اما در [۶] کاری که انجام شده به این شکل است که منطق‌های LTL و CTL با عبارات منظم [۱۴] جایگزین شده‌اند. علت این کار دو نکته بوده، اولی اینکه استفاده از عبارات منظم به جای منطق‌های نام برده شده می‌تواند برای برنامه نویسان ساده‌تر باشد و دومی اینکه عبارت منظم از منطق‌های نام برده شده قدرت بیان بالاتری دارد. [۲۲] در ادامه ی کار واریسی مدل با استفاده از موجودات جدید تعریف شده به سه شکل مختلف بیان شده. در هر مرتبه بیان واریسی مدل، به‌گفته‌ی نویسنده، ”ساختارمند” تر شده. می‌توان دریافت که فایده ساختارمندتر بودن بیان این است که پیاده‌سازی راحت‌تری دارد. حال به بیان و بررسی تعاریف و خواص آن‌ها می‌پردازیم.

۱.۲ نحو زبان مورد بررسی

زبان بیان برنامه‌ها زیرمجموعه‌ای از دستورات زبان C است، به شکل زیر:

$$x, y, \dots \in \mathcal{X}$$

$$A \in \mathcal{A} ::= 1 \mid x \mid A_1 - A_2$$

$$B \in \mathcal{B} ::= A_1 < A_2 \mid B_1 \text{ nand } B_2$$

$$E \in \mathcal{E} ::= A \mid B$$

$$S \in \mathcal{S} ::=$$

$$\begin{aligned}
& x = A; \\
& | \ ; \\
& | \text{ if } (B) \ S \ | \text{ if } (B) \ S \text{ else } S \\
& | \text{ while } (B) \ S \ | \text{ break}; \\
& | \{SI\} \\
& SI \in \mathfrak{S} \mathfrak{L} ::= SI \ S \ | \ \epsilon \\
& P \in \mathfrak{P} ::= SI
\end{aligned}$$

در اینجا زیر مجموعه‌ای از دستورات زبان C را داریم. همین‌طور که قابل مشاهده است این زبان تا حد ممکن کوچک شده. علت این کار را بعداً عمیق‌تر حس خواهیم کرد. علت ساده‌تر شدن کار برای ارائه‌ی معنانشناسی و تعبیر مجرد آن است. در اینجا راحتی آن برنامه‌نویسی که قرار است با این زبان برنامه بنویسد مطرح نبوده چون اصلاً این زبان برای این کار ساخته نشده. نویسنده‌ی [۶] در اینجا صرفاً می‌خواسته فرآیند را نشان دهد. اگر به فرض برای زبانی مانند پایتون بخواهیم درستی‌یابی با استفاده از روش ارائه شده را درست کنیم، می‌توانیم همه‌ی راهی که در [۶] برای زبان توصیف شده، رفته‌شده را برای پایتون هم برویم و به یک تحلیل‌گر ایستا برای پایتون برسیم. در مورد قدرت بیان این زبان هم می‌توانیم بگوییم که می‌توانیم باقی اعداد را از روی عدد ۱ و عملگر منها بسازیم. مثلاً ابتدا ۰ را به کمک ۱-۱ می‌سازیم و سپس با استفاده از ۰ می‌توانیم یکی یکی اعداد منفی را بسازیم و سپس بعد از آن به سراغ اعداد مثبت می‌رویم که با کمک ۰ و هر عدد منفی‌ای که ساختیم، ساخته می‌شوند. باقی اعداد و حتی باقی عملگرها (یعنی به غیر از اعداد طبیعی) نیز از روی آنچه داریم قابل ساختن است. در مورد عبارت‌های بولی نیز داستان به همین منوال است. یعنی اینجا صرفاً ادات شفر تعریف شده و باقی عملگرهای بولی را می‌توان با استفاده از همین عملگر ساخت. باقی دستورات نیز دستورات شرط و حلقه هستند. باقی دستورات قرار است مطابق چیزی که از زبان C انتظار داریم کار بکنند. در مورد دستور break ذکر این نکته ضروری است که اجرای آن قرار است اجرای برنامه را از دستوری بعد از داخلی‌ترین حلقه‌ای که break داخلش قرار دارد ادامه دهد. در پایان می‌توان ثابت کرد که این زبان هم قدرت با مدل دیویس [۹] است. توجه داریم که هرچه در این بخش در مورد معنای دستورات این زبان گفتیم، به هیچ وجه صوری نبود و صرفاً درک شهودی‌ای که از معنای اجرای هریک از دستورات داشتیم را بیان کردیم. بیان صوری معنای برنامه‌ها را، که برخلاف درک شهودی‌مان قابل انتقال به کامپیوتر است، در ادامه بیان خواهیم کرد. طبیعتاً این بیان صوری از روی یک درک شهودی ساخته شده‌است.

۲.۲ معنانشناسی زبان مورد بررسی

معنانشناسی زبانی را که در بخش پیش آوردیم با کمک مفاهیمی به نام برچسب و رد پیشوندی و عملگر چسباندن روی دو رود پیشوندی مختلف تعریف خواهیم کرد و نام این معنانشناسی نیز معنانشناسی رد پیشوندی است.

۱.۲.۲ برچسب‌ها

باوجود اینکه خود زبان C در قسمتی از زبان خود چیزهایی به نام برچسب دارد اما همین‌طور که در بخش پیشین دیدیم، در زبانی که اینجا در حال بحث روی آن هستیم خبری از برچسب‌ها نیست. اما برای تعریف صوری معنای برنامه‌ها، به شکلی که مورد بحث است، به آن‌ها نیاز است. در این بخش ابتدا به توضیح مختصر در مورد برچسب‌ها در معنانشناسی زبان مورد بحث می‌پردازیم. تعاریف صوری دقیق این موجودات در پیوست [۶] آورده شده اند. برای اختصار از آوردن مستقیم این تعاریف در اینجا خودداری کرده‌ایم. در زبانمان Sها بخشی از موجودات موجود در زبان هستند. برچسب‌ها را برای Sها تعریف می‌کنیم. برچسب‌ها با کمک توابع lab, in, brks-of, brk-to, esc, aft, at تعریف می‌شوند. درواقع هر S به ازای بعضی از این توابع یک برچسب دارد و این‌ها درواقع نشان‌دهنده‌ی آن برچسب هستند. بعضی دیگر این توابع برای هر S ممکن است یک مجموعه از برچسب‌ها را تعیین کند و یکی از آن‌ها هم با گرفتن S یک مقدار بولی را برمی‌گرداند.

at[S] : برچسب شروع S

aft[S] : برچسب پایان S، اگر پایانی داشته باشد

esc[S] : یک مقدار بولی را باز می‌گرداند که بسته به اینکه در S دستور break وجود دارد یا خیر، مقدار درست یا غلط را برمی‌گرداند.

brk-to[S] : برچسبی است که اگر حین S دستور break اجرا شود، برنامه از آن نقطه ادامه پیدا می‌کند.

brks-of[S] : مجموعه‌ای از برچسب break های S را برمی‌گرداند.

in[S] : مجموعه‌ای از تمام برچسب‌های درون S را برمی‌گرداند.

labs[S] : مجموعه‌ای از تمام برچسب‌هایی که با اجرای S قابل دسترسی هستند را برمی‌گرداند.

۲.۲.۲ رد پیشوندی

پس از تعریف برچسب‌ها به سراغ تعریف رد پیشوندی می‌رویم. هر رد پیشوندی یک دنباله است که قرار است توصیفی از چگونگی اجرای برنامه باشد. اعضای دنباله وضعیت‌ها هستند. یک وضعیت، یک زوج مرتب $\langle l, \rho \rangle$ است. که در آن l یک برچسب و ρ یک محیط است. یک محیط تابعی است از مجموعه متغیرها \mathbb{X} به مجموعه مقادیر \mathbb{V} و مجموعه خود وضعیت‌ها را هم با \mathbb{S} نشان می‌دهیم. مجموعه محیط‌ها را نیز با \mathbb{EV} نشان می‌دهیم. وضعیت‌ها همان‌طور که از نامشان پیداست قرار است موقعیت لحظه‌ای برنامه را توصیف کنند. l قرار است برچسب برنامه‌ی در حال اجرا باشد و ρ مقدار متغیرها را در آن موقع از اجرای برنامه نشان می‌دهد. دنباله‌های ما می‌توانند متناهی یا نامتناهی باشند. مجموعه ردهای پیشوندی متناهی را با \mathbb{S}^+ و مجموعه ردهای پیشوندی نامتناهی را با \mathbb{S}^∞ نمایش می‌دهیم. مجموعه همه ردهای پیشوندی را هم با $\mathbb{S}^{+\infty}$ نمایش می‌دهیم. باتوجه به آنچه گفتیم، یک عملگر چسباندن \bowtie را روی ردهای پیشوندی تعریف می‌کنیم. اگر داشته باشیم $\sigma_1, \sigma_2 \in \mathbb{S}$, $\pi_1, \pi_2 \in \mathbb{S}^{+\infty}$ داریم:

$$\begin{array}{ll} \pi_1 \bowtie \pi_2 = \pi_1 & \text{اگر } \pi_1 \in \mathbb{S}^+ \text{ داریم} \\ \pi_1 \bowtie \pi_2 & \text{اگر } \sigma_1 \neq \sigma_2 \\ \pi_1 \sigma_1 \bowtie \sigma_1 \pi_2 = \pi_1 \sigma_1 \pi_2 & \text{اگر } \pi_1 \in \mathbb{S}^\infty \text{ داریم} \end{array}$$

همین‌طور \exists هم یک رد پیشوندی است که حاوی هیچ وضعیتی نیست. به عبارت دیگر یک دنباله‌ی تهی است.

۳.۲.۲ تعریف صوری معناسازی رد پیشوندی

در این بخش قرار است دو تابع A و B را به ترتیب روی عبارات حسابی و بولی زبانمان یعنی A ها و B ها تعریف کنیم سپس با کمک آنها \mathcal{S}^* را روی مجموعه‌ای از اجتماع معنای S ها و \mathcal{S} ها تعریف می‌کنیم. پس در نهایت هدف ما تعریف \mathcal{S}^* است. تعریف A به این شکل است:

$$\begin{aligned} A[1]\rho &= 1 \\ A[x]\rho &= \rho(x) \\ A[A_1 - A_2]\rho &= A[A_1]\rho - A[A_2]\rho \end{aligned}$$

تعریف B نیز به شکل زیر است:

$$\begin{aligned} B[A_1 < A_2]\rho &= \text{True} & \text{اگر } A[A_1]\rho \text{ کوچکتر از } A[A_2]\rho \text{ باشد} \\ B[A_1 < A_2]\rho &= \text{False} & \text{اگر } A[A_1]\rho \text{ بزرگتر از } A[A_2]\rho \text{ باشد} \\ B[B_1 \text{ and } B_2]\rho &= \neg (B[B_1]\rho \wedge B[B_2]\rho) \end{aligned}$$

که طبعاً \wedge و \neg در فرازبان هستند.

در ادامه به تعریف S^* می‌پردازیم. این کار را با تعریف S^* روی هر ساخت S و SI انجام می‌دهیم. پیش از ادامه‌ی بحث باید این نکته را درمورد علامت‌گذاری‌هایمان ذکر کنیم که منظور از $S ::= lbreak;$ این است که تاکید کرده‌ایم که S با برجسب l شروع شده‌است وگرنه همین طور که گفتیم l جزو زبان نیست.

اگر $S ::= break;$ باشد، ردهای پیشوندی متناظر با اجرای این دستور را به شکل مجموعه‌ی زیر تعریف می‌کنیم:

$$S^*[S] = \{\langle at[S], \rho \rangle | \rho \in \mathbb{EV}\} \cup \{\langle at[S], \rho \rangle \langle brk - to[S], \rho \rangle | \rho \in \mathbb{EV}\}$$

اگر $S ::= x = A;$ باشد، ردهای پیشوندی متناظر با اجرای این دستور را به شکل مجموعه‌ی زیر تعریف می‌کنیم:

$$S^*[S] = \{\langle at[S], \rho \rangle | \rho \in \mathbb{EV}\} \cup \{\langle at[S], \rho \rangle \langle aft[S], \rho[x \leftarrow \mathcal{A}[A]\rho] \rangle | \rho \in \mathbb{EV}\}$$

اگر $S ::= if(B)S_t$ باشد، ردهای پیشوندی متناظر با اجرای این دستور را به شکل مجموعه‌ی زیر تعریف می‌کنیم:

$$S^*[S] = \{\langle at[S], \rho \rangle | \rho \in \mathbb{EV}\} \cup \{\langle at[S], \rho \rangle \langle aft[S], \rho \rangle | \mathcal{B}[B]\rho = False\}$$

$$\cup \{\langle at[S], \rho \rangle \langle at[S_t], \rho \rangle \pi | \mathcal{B}[B]\rho = True \wedge \langle at[S_t], \rho \rangle \pi \in \mathcal{S}[S_t]\}$$

اگر $S ::= if(B)S_t else S_f$ باشد، ردهای پیشوندی متناظر با اجرای این دستور را به شکل مجموعه‌ی زیر تعریف می‌کنیم:

$$\mathcal{S}[S] = \{\langle at[S], \rho \rangle | \rho \in \mathbb{EV}\}$$

$$\cup \{\langle at[S], \rho \rangle \langle at[S_f], \rho \rangle \pi | \mathcal{B}[B]\rho = False \wedge \langle at[S_f], \rho \rangle \pi \in \mathcal{S}[S_f]\}$$

$$\cup \{\langle at[S], \rho \rangle \langle at[S_t], \rho \rangle \pi | \mathcal{B}[B]\rho = True \wedge \langle at[S_t], \rho \rangle \pi \in \mathcal{S}[S_t]\}$$

اگر $SI ::= \exists$ باشد، ردهای پیشوندی متناظر با اجرای این دستور را به شکل مجموعه‌ی زیر تعریف می‌کنیم:

$$\mathcal{S}[SI] = \{\langle at[SI], \rho \rangle | \rho \in \mathbb{EV}\}$$

اگر $S ::= S'$ باشد، ردهای پیشوندی متناظر با اجرای این دستور را به شکل مجموعه‌ی زیر تعریف می‌کنیم:

$$\mathcal{S}[S] = \mathcal{S}[S'] \cup (\mathcal{S}[S'] \bowtie \mathcal{S}[S])$$

اگر $S ::= \text{while}(B)S_b$ باشد، ماجرا نسبت به حالات قبل اندکی پیچیده‌تر می‌شود. تابعی به اسم \mathcal{F} را تعریف خواهیم‌کرد که در حقیقت دو ورودی دارد. ورودی اول آن یک دستور حلقه است و ورودی دوم آن یک مجموعه. به عبارتی دیگر می‌توانیم بگوییم به ازای هر حلقه یک تابع \mathcal{F} جداگانه تعریف می‌شود که مجموعه‌ای از ردهای پیشوندی را می‌گیرد و مجموعه‌ای دیگر از همین موجودات را باز می‌گرداند. کاری که این تابع قرار است انجام دهد این است که انگار یک دور دستورات داخل حلقه را اجرا می‌کند و دنباله‌هایی جدید را از دنباله‌های قبلی می‌سازد. معنای یک حلقه را کوچکترین نقطه ثابت این تابع در نظر می‌گیریم. در ادامه تعریف \mathcal{F} آمده. با دیدن تعریف می‌توان به دلیل این کار پی‌برد. آن نقطه‌ای که دیگر \mathcal{F} روی آن اثر نمی‌کند یا حالتی است که در آن دیگر شرط حلقه برقرار نیست و اصولاً قرار نیست دستورات داخل حلقه اجرا شوند که طبق تعریف \mathcal{F} می‌توانیم ببینیم که \mathcal{F} در این حالت چیزی به ردهای پیشوندی اضافه نمی‌کند. یا اینکه حلقه به دستور break خورده که در آن صورت وضعیتی به ته ردهای پیشوندی اضافه می‌شود که برچسبش خارج از مجموعه برچسب دستورات حلقه است و همین اضافه کردن هر چیزی را به ته ردهای پیشوندی موجود، توسط \mathcal{F} غیرممکن می‌کند. بنابراین نقطه ثابت مفهوم مناسبی است برای اینکه از آن در تعریف صوری معنای حلقه استفاده کنیم. علت اینکه کوچکترین نقطه ثابت را به عنوان معنای حلقه در نظر می‌گیریم هم این است که مطمئن هستیم کوچکترین نقطه ثابت، هر رد پیشوندی ای را در خود داشته باشد به معنای اجرای برنامه مرتبط است. برای درک بهتر این نکته می‌توان به این نکته توجه کرد که با اضافه کردن وضعیت‌هایی کاملاً بی‌ربط به اجرای برنامه به ته ردهای پیشوندی، که صرفاً برچسب متفاوتی با آخرین وضعیت هر رد پیشوندی دارند، نقطه ثابت جدیدی ساخته ایم. پس اگر خودمان را محدود به انتخاب کوچکترین نقطه ثابت نکنیم، به توصیفات صوری خوبی از برنامه‌ها دست پیدا نخواهیم‌کرد. در مورد نقطه ثابت تنها این نکته باقی می‌ماند که اصلاً از کجا می‌دانیم که چنین نقطه ثابتی وجود دارد که در این صورت باید گفت مجموعه‌هایی که از ردهای پیشوندی تشکیل می‌شوند با عملگر زیرمجموعه بودن یک شبکه را تشکیل می‌دهند و بنا به قضیه تارسکی [۲۰] برای چنین موجودی نقطه ثابت وجود دارد. تعاریف موجوداتی که درموردشان صحبت کردیم به این شکل است:

$$\mathcal{S}[S] = \text{lfp}^{\subseteq} \mathcal{F}[S]$$

$$\mathcal{F}[S]X = \{ \langle \text{at}[S], \rho \rangle \mid \rho \in \mathbb{E}\mathbb{V} \} \cup$$

$$\{ \pi_2 \langle l, \rho \rangle \langle \text{aft}[S], \rho \rangle \mid \pi_2 \langle l, \rho \rangle \in X \wedge \mathcal{B}[B]\rho = \text{False} \wedge l = \text{at}[S] \} \cup$$

$$\{ \pi_2 \langle l, \rho \rangle \langle \text{at}[S_b], \rho \rangle \mid \pi_2 \langle l, \rho \rangle \in X \wedge \mathcal{B}[B]\rho = \text{True} \wedge$$

$$\langle at[S_b], \rho \rangle \pi_3 \in \mathcal{S}[S_b] \wedge l = at[S]\}$$

اگر $S ::= S$ باشد، ردهای پیشوندی متناظر با اجرای این دستور را به شکل مجموعه‌ی زیر تعریف می‌کنیم:

$$\mathcal{S}[S] = \{\langle at[S], \rho \rangle | \rho \in \mathbb{EV}\} \cup \{\langle at[S], \rho \rangle \langle aft[S], \rho \rangle | \rho \in \mathbb{EV}\}$$

اگر $S ::= \{S\}$ باشد، ردهای پیشوندی متناظر با اجرای این دستور را به شکل مجموعه‌ی زیر تعریف می‌کنیم:

$$\mathcal{S}[S] = \mathcal{S}[S]$$

۳.۲ ویژگی‌های معنایی برنامه‌ها

تا به اینجای کار یک زبان آورده‌ایم و برای آن معنا تعریف کرده‌ایم. در این فصل می‌خواهیم در مورد ویژگی‌های برنامه‌هایی که در این زبان نوشته می‌شوند با توجه به معنای صوری‌ای که تعریف کرده‌ایم، صحبت کنیم. دقت شود که برای برنامه‌هایی که در یک زبان برنامه‌نویسی نوشته می‌شوند می‌توان به اشکال مختلفی ویژگی تعریف کرد؛ مثلاً ویژگی‌های نحوی، مثل اینکه طول برنامه چند خط است یا هر کاراکتر چند بار به کار رفته، یا ویژگی‌های محاسباتی، مثل بررسی سرعت برنامه یا میزان استفاده‌ی آن از حافظه که عموماً در نظریه الگوریتم و پیچیدگی محاسبات بررسی می‌شود. منظور ما در اینجا از تعریف ویژگی، متناسب است با معناسازی‌ای که برای برنامه‌هایمان تعریف کرده‌ایم. معناسازی‌ای که تعریف کرده‌ایم در واقع سیر محاسباتی برنامه را توصیف می‌کند و ما می‌خواهیم ویژگی‌ها را با توجه به این موضوع تعریف کنیم. در این صورت می‌توانیم صحت عملکرد برنامه‌ها را با توجه به صادق بودن ویژگی‌هایی که در مورد آن‌ها تعریف شده بفهمیم. ابتدا به تعریف ویژگی‌ها می‌پردازیم، سپس به سراغ تعریف یک نوع عبارت منظم می‌رویم که از آن برای بیان ویژگی‌ها استفاده می‌شود.

۱.۳.۲ ویژگی‌های معنایی

همان‌طور که در بخش قبلی دیدیم، معنای هر برنامه با یک مجموعه‌ی $\mathcal{S}^*[S]$ مشخص می‌شود. وقتی می‌خواهیم ویژگی‌هایی را برای موجوداتی که به کمک مجموعه‌ها تعریف شده اند بیان کنیم، اینکه ویژگی‌ها را هم با مجموعه‌ها بیان کنیم کار معقولی به نظر می‌رسد. مثل اینکه بخواهیم ویژگی

زوج بودن را در مورد اعداد طبیعی بیان کنیم. می‌توانیم مجموعه‌ی \mathbb{E} را به عنوان مجموعه‌ی همه‌ی اعداد زوج در نظر بگیریم و اینکه یک عدد زوج هست یا نه را عضویتش در مجموعه‌ی \mathbb{E} تعریف کنیم. پس یعنی در مورد اعداد طبیعی قرار است هر ویژگی به شکل زیرمجموعه‌ای از تمام این اعداد در نظر گرفته شود. یعنی هر عضو $P(\mathbb{N})$ بنا به تعریف ما یک ویژگی از اعداد طبیعی است. در مورد برنامه‌ها نیز قرار است همین رویه را پیش بگیریم. تابع S^* از نوع $P(S^+) \rightarrow \mathfrak{P}$ است. یعنی یک برنامه را در ورودی می‌گیرد و یک مجموعه از ردهای پیشوندی را باز می‌گرداند. پس می‌توانیم هر ویژگی را به عنوان زیرمجموعه‌ای از $P(S^+)$ تعریف کنیم، به عبارت دیگر عضوی از $P(P(S^+))$.

۲.۳.۲ عبارات منظم

در اینجا توصیف ویژگی‌ها برای هر برنامه باید یک چارچوب داشته باشد. در صورت قدیمی روش واریسی مدل ما از منطق‌های زمانی برای بیان ویژگی‌ها به صورت صوری استفاده می‌کردیم و این احتیاج به یک زبان برای صوری کردن کامل کار را، که رسیدن به بیان مسئله‌ی واریسی مدل است، به ما نشان می‌دهد. در اینجا ما با داستان دیگری هم رو به رو هستیم و آن این است که از آنجایی که با مجموعه‌ها سر و کار داریم و مجموعه‌ها چندان موجودات ساختنی‌ای نیستند (برخلاف مدل کریپکی)، بهتر است یک موجود ساختنی مثل یک زبان صوری برای بیان آن‌ها داشته باشیم. در این فصل قصد داریم یک نوع عبارت منظم را برای این منظور تعریف کنیم. ابتدا زبان این عبارت منظم را تعریف می‌کنیم، سپس به سراغ معناسازی آن می‌رویم.

۳.۳.۲ زبان عبارات منظم

فرق عمده‌ای که زبان عبارات منظم ما با عبارات منظم کلاسیک دارد در کاراکترهاست. کاراکترها در زبان کلاسیک موجوداتی اتمی بودند، اما در اینجا ساختار دارند. در اینجا به جای هر کاراکتر یک زوج متشکل از مجموعه‌ی L و عبارت بولی B تشکیل شده‌اند که این زوج را به شکل $L : B$ در زبانمان نمایش می‌دهیم. زبان ما به شکل BNF زیر است:

$$L \in P(\mathbb{L})$$

$$x, y, \dots \in \mathbb{X}$$

$$\underline{x}, \underline{y}, \dots \in \underline{\mathbb{X}}$$

$$B \in \mathbb{B}$$

$$R \in \mathbb{R}$$

$$\begin{aligned}
R ::= & \varepsilon \\
& | L : B \\
& | R_1 R_2 \text{ (or } R_1 \bullet R_2) \\
& | R_1 \mid R_2 \text{ (or } R_1 + R_2) \\
& | R_1^* \\
& | R_1^+ \\
& | (R_1)
\end{aligned}$$

همان طور که قابل مشاهده است در اینجا عملگرهای دوتایی چسباندن (•) و انتخاب (|) را داریم، به همراه عملگرهای یگانی * و +. در ادامه خواهیم دید که در فرازبان معنی عملگر یگانی + به وسیله‌ی عملگر یگانی دیگر قابل بیان است، هرچند که در زبانمان هم برای سهولت کار از بیان این عملگر اجتناب نشده. توجه شود که پرانتزها هم جزئی از زبان قرار داده شده‌اند. همین طور در اینجا می‌خواهیم از تعدادی عبارات مخفف که در ادامه کارمان را راحت تر می‌کنند صحبت کنیم. منظور از زوج B : ? همان B : \mathbb{L} است. عبارت B : l به جای عبارت B : {l} به کار می‌رود و منظور از عبارت B : l- نیز عبارت B : $\mathbb{L} \setminus \{l\}$ است. با یک نگاه به دستور این زبان یک نکته‌ی چشمگیر برای ما، با توجه به موجوداتی که در بخش قبل تعریف کردیم، با نگاه به قواعد این زبان می‌تواند وجود یک مجموعه‌ی \mathbb{X} در کنار \mathbb{X} که از قبل داشتیم باشد. قرار است به ازای هر $x \in \mathbb{X}$ یک $\underline{x} \in \mathbb{X}$ داشته باشیم. منظور از \underline{x} مقدار متغیر x در ابتدای هر برنامه است. این یعنی تابع $\rho : \mathbb{X} \rightarrow \mathbb{V}$ که \mathbb{V} مجموعه‌ی مقادیر متغیرهاست (در بخش قبل به این اشاره نشد اما خود ρ هایی که در بخش قبل داشتیم هم از نوع $\mathbb{X} \rightarrow \mathbb{V}$ بود. با توجه به زبانمان و توضیحاتی که در گذشته دادیم، می‌توان در نظر گرفت که در اینجا \mathbb{V} همان \mathbb{Z} یعنی اعداد صحیح است. دلیل استفاده از نماد \mathbb{V} به عنوان مجموعه مقادیر این است که روح کلی نگرانه‌ی کار حفظ شود). همان طور که پیش تر گفتیم برای اشاره به یک تابع ρ از کلمه‌ی ”محیط“ استفاده می‌شود. به همین منوال در ادامه برای اشاره به ρ از ”محیط اولیه“ استفاده می‌کنیم. برای اشاره به مجموعه‌ی همه‌ی محیط‌های اولیه هم از نماد $\mathbb{E}\mathbb{V}$ استفاده می‌کنیم. بقیه‌ی موجودات از جمله برچسب‌ها و عبارات بولی را هم که قبلاً داشتیم. در ادامه به بیان صوری معنای زبان بیان شده می‌پردازیم. پس از آن می‌توانیم با بررسی چند مثال، از اینکه معنای هر عبارت منظم چیست درکی شهودی به دست آوریم.

۴.۳.۲ معناسازی عبارات منظم

معنای عبارات منظم را با استفاده از تابع S^r نشان می‌دهیم. این تابع به این شکل تعریف می‌شود که در ورودی یک عبارت منظم R را می‌گیرد، سپس یک مجموعه از زوج مرتب‌های (یا همان‌طور که پیش‌تر نام‌گذاری کردیم "حالت‌ها") $\langle \rho, \pi \rangle$ را که $\rho \in \underline{\mathbb{EV}}$ و $\pi \in S^*$ باز می‌گرداند. بنابراین این تابع از نوع $\mathbb{R} \rightarrow P(\underline{\mathbb{EV}} \times S^*)$ است. همین‌طور دقت شود که تا به حال از S^* صحبتی نکرده بودیم و فقط S^+ را معرفی کرده بودیم. S^* نیز برابر است با $S^+ \cup \{\epsilon\}$ (به لحاظ معنایی همان عملگر $*$ است که در زبان عبارات منظم هم هست، مشهور به ستاره‌ی کلینی).
تعریف استقرایی تابع S^r به شکل زیر است:

$$S^r[\epsilon] = \{\langle \rho, \epsilon \rangle \mid \rho \in \underline{\mathbb{EV}}\}$$

[یعنی معنای عبارت منظم ϵ مجموعه‌ای شامل حالت‌هایی از محیط‌های اولیه‌ی مختلف در کنار رد پیشوندی تهی استفاده می‌کند.]

$$S^r[L : B] = \{\langle \rho, \langle l, \rho \rangle \rangle \mid l \in L \wedge B[B]\rho, \rho\}$$

[این یعنی معنای عبارت $S^r[L : B]$ حالت‌هایی هستند که عضو اول آن‌ها محیط‌های اولیه مختلف هستند (مانند مورد قبلی و البته در موارد آتی!) و عضو دوم آن‌ها ردهای پیشوندی تک‌عضوی $\langle l, \rho \rangle$ هستند که در آن‌ها برچسب l باید در L که مجموعه‌ای از برچسب‌هاست حضور داشته باشد و عبارت بولی B باید درباره‌ی محیط اولیه ρ و محیط ρ برقرار باشد. حتماً متوجه این نکته شدید که B در اینجا به جای اینکه از نوع $\underline{\mathbb{EV}} \rightarrow \mathbb{BOOL}$ باشد، همان‌طور که قبلاً تعریف کردیم، از نوع $\underline{\mathbb{EV}} \rightarrow \underline{\mathbb{EV}} \rightarrow \mathbb{BOOL}$ است. (منظور از \mathbb{BOOL} همان مجموعه‌ی $\{True, False\}$ است.) در اینجا A و B را در ادامه با نوع‌های متفاوت دوباره تعریف خواهیم کرد، که البته فرق اساسی‌ای با تعریف قبلی ندارد و صرفاً گسترشی ساده از آن است.]

$$S^r[R_1 R_2] = S^r[R_1] \bowtie S^r[R_2]$$

به‌طوری که در آن برای هر دو مجموعه‌ی S و S' از ردهای پیشوندی:

$$S \bowtie S' = \{\langle \rho, \pi \pi' \rangle \mid \langle \rho, \pi \rangle \in S \wedge \langle \rho, \pi' \rangle \in S'\}$$

[این یعنی اگر یک عبارت منظم داشته باشیم که از چسباندن R_1 و R_2 به هم ساخته شده باشد، آنگاه معنای این عبارت منظم با چسباندن ردهای پیشوندی موجود در مولفه‌ی دوم زوج مرتب‌هایی که اعضای مجموعه‌ی معنای این دو عبارت منظم هستند و گذاشتن این رد پیشوندی‌های حاصل از چسباندن در معنای عبارت منظم جدید تعریف می‌شود.
تا این تکه از تعریف معنای عبارت منظم که رسیده‌ایم می‌توانیم تا حدی به دستیابی به درکی شهودی

از اینکه به چه نحوی قرار است عبارات منظم راهی برای توصیف ویژگی در مورد برنامه‌ها باشد نزدیک‌تر شویم. همان‌طور که در مورد قبل دیدیم هر زوج $L : B$ دقیقا به یک حالت داخل یک رد پیشوندی اشاره می‌کند. انگار که قرار است این زوج‌ها موازی با حالت‌ها در ردهای پیشوندی موجود در معنای یک برنامه جلو روند و منطبق باشند تا واریسی مدل انجام شود. درک این موضوع اولین قدم ماست در دیدن عصاره‌ی روش واریسی مدل در ادبیاتی که از اول این فصل علم کرده‌ایم.

فصل ۳

وارسی مدل منظم

در این فصل قرار است بیانی ساختارمندتر از روش واریسی مدل داشته باشیم. اهمیت ساختارمند تر بودن در این است که بیانی که در فصل پیش داشتیم تا پیاده سازی فاصله‌ی بسیاری دارد، چون همان‌طور که پیش‌تر گفته شد مجموعه‌ها موجودات ساختنی‌ای نیستند و کار با آن‌ها حین نوشتن برنامه‌ای کامپیوتری که قرار است پیاده‌سازی روش مورد بحث ما باشد را سخت می‌کند.

فصل ۴

وارسی مدل ساختارمند

فصل ۵

نتیجه گیری

واژه‌نامه فارسی به انگلیسی

واژه‌نامه انگلیسی به فارسی

Bibliography

- [1] Committee to review chinook zd 576 crash. report from the select committee on chinook zd 576., Feb 2002.
- [2] A. S. E. Al. Mars climate orbiter mishap investigation board phase i report., November 1999.
- [3] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to Coq Proof Assistant*. MIT Press, 2022.
- [4] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.
- [5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, London, Cambridge, 1999.
- [6] P. Cousot. Calculational design of a regular model checker by abstract interpretation. In R. M. Hierons and M. Mosbah, editors, *ICTAC*, volume 11884 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2019.
- [7] P. Cousot. *Principals of Abstract Interpretation*. MIT Press, 2021.
- [8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.
- [9] M. Davis and E. Weyuker. *Computability, Complexity, and Languages*. Academic Press, New York, 1983.

- [10] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of philosophical logic*, pages 99–217. Springer, 2001.
- [11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [12] M. Huth and M. Ryan. *Logic in computer science : modelling and reasoning about systems*. Cambridge University Press, Cambridge [U.K.]; New York, 2004.
- [13] X. R. K. Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, 2020.
- [14] S. Kleene. Representation of Events in Nerve Nets and Finite Automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, 1956.
- [15] S. A. Kripke. A completeness theorem in modal logic¹. *The journal of symbolic logic*, 24(1):1–14, 1959.
- [16] J. Lions. Ariane 5 Flight 501 Failure: Report of the Inquiry Board, July 1996.
- [17] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, Hoboken and N.J, 3rd ed edition, 2012.
- [18] B. C. Pierce, A. Azevedo de Amorim and Chris Casinghino, M. Gaboardi, M. Greenberg, C. Hrițcu, V. Sjöberg, A. Tolmach, and B. Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018.
- [19] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.
- [20] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.
- [21] G. Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.
- [22] P. Wolper. Temporal logic can be more expressive. *Inf. Control.*, 56(1/2):72–99, January/February 1983.

Abstract

Abstract goes here...



College of Science
School of Mathematics, Statistics, and Computer Science

Thesis Title

Author name

Supervisor: name
Co-Supervisor: name
Advisor: name

A thesis submitted to Graduate Studies Office
in partial fulfillment of the requirements for the degree of
B.Sc./Master of Science/Doctor of Philosophy in
Pure Mathematics/ Applied Mathematics/ Statistics/ Computer Science

yyyy